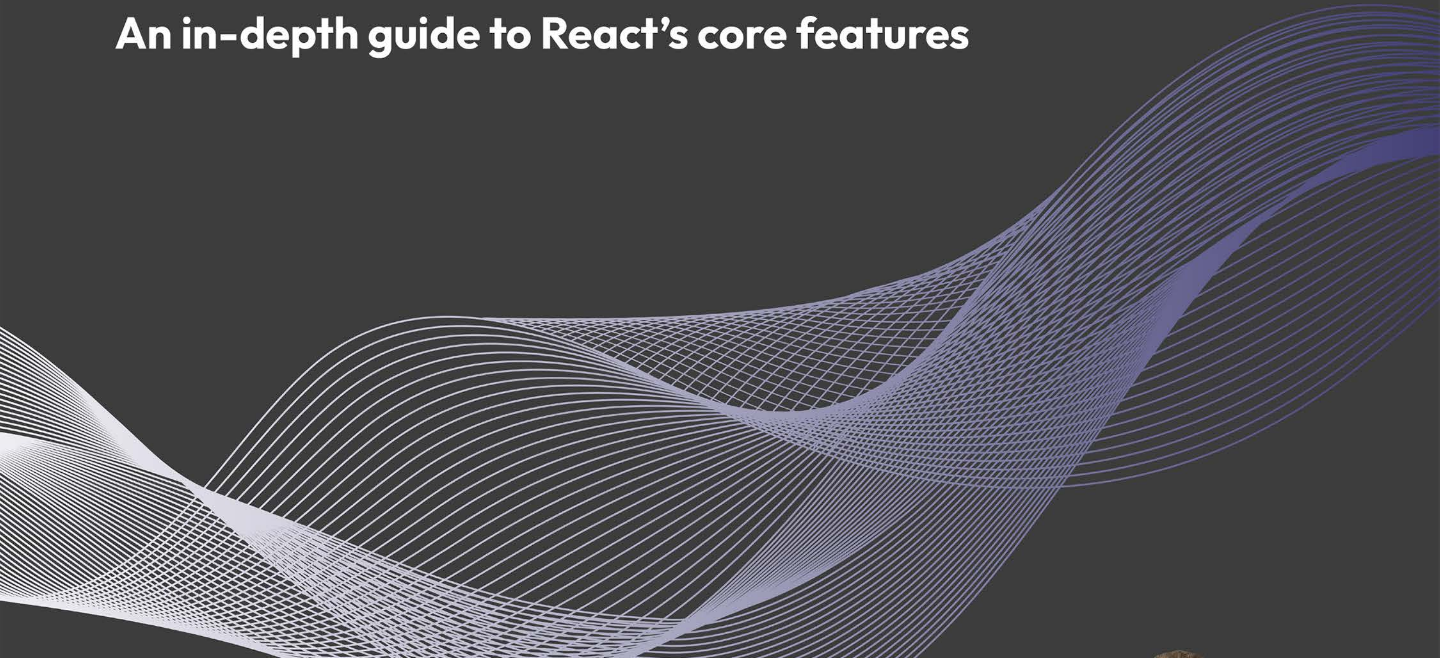
EXPERT INSIGHT

React
Key Concepts

An in-depth guide to React's core features



Second Edition

Maximilian Schwarzmüller

<packt>

React Key Concepts

Second Edition

An in-depth guide to React's core features

Maximilian Schwarzmüller

packt

React Key Concepts

Second Edition

Copyright © 2024 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Publishing Product Manager: Lucy Wan
Acquisition Editor – Peer Reviews: Jane Dsouza
Project Editor: Janice Gonsalves
Senior Development Editor: Elliot Dallow
Copy Editor: Safis Editing
Technical Editor: Tejas Mhasvekar
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Presentation Designer: Ajay Patule
Developer Relations Marketing Executive: Priyadarshini Sharma

First published: December 2022
Second edition: December 2024

Production reference: 1231224

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul’s Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83620-227-1

`www.packt.com`

Contributors

About the Author

Maximilian Schwarzmüller is a professional web developer and bestselling online course instructor. On Udemy, he is one of the most popular and biggest online instructors, teaching more than 3 million students worldwide. Students can become developers by exploring his more than 40 courses, most of them bestsellers in their respective categories.

Besides helping students from all over the world, Maximilian loves exploring and mastering new technologies, building exciting digital products, and sharing his knowledge with fellow developers. He's driven by his passion for good code and engaging websites and apps.

I may be the author of this book but planning, polishing, and publishing this book was really a group effort.

Most of all, I'm thankful for all the support from my wife Anna-Maria. You're the love of my life!

I also want to thank my publisher, Packt: Thank you Bridget, Megan, Elliot, Janice, Lucy, Tejas, and everyone else who was involved!

About the Reviewers

Cihan Yakar has over twenty years of experience in software development. He specializes in fullstack development and machine learning, creating applications with .NET and Node.js. An enthusiastic learner and knowledge sharer, Cihan often speaks at user group meetings. He is the founder of Bitsody Software and Defne Software. He was also a technical reviewer of *The TypeScript Workshop*. To discover more about his professional journey, feel free to connect with him on LinkedIn. When not working, Cihan enjoys spending time with his family and indulging in his passion for all things Star Trek.

Slava Knyazev has been writing software since his early teenage years and is always seeking to find ways to improve his mastery of the craft. He has worked for well-known names, including theScore, Amazon Web Services, and Airbnb. When he isn't writing code, he dives into technical topics on his blog, *Building Better Software Slower*.

Eric Harvey is a consultant for Enwise Webtech LLC, focused on EdTech and secure systems integrations. He has worked in technology since 1998, his roles have included: applications engineer, web developer, manager of learning systems at a major university, and solutions engineer. In 2005, he founded a web development and hosting services company. Outside of work he is an avid collector of board games and vintage computers, and plays mandolin in a local Celtic string band.

I would like to thank my kids – Amber, Nate, and Rylan – and my wife, Meredith, for being understanding, patient, and always loving.

Join Us on Discord

Read this book alongside other users, AI experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the author via Ask Me Anything sessions, and much more.

Scan the QR code or visit the link to join the community.

`https://packt.link/ReactKeyConcepts2e`

Table of Contents

Preface xix

Chapter 1: React – What and Why 1

Introduction 1
What is React? 2
The Problem with "Vanilla JavaScript" 2
React and Declarative Code 6
  How React Manipulates the DOM • 9
Introducing SPAs 10
  Creating a React Project with Vite • 11
Summary and Key Takeaways 13
  What's Next? • 14
  Test Your Knowledge! • 14

Chapter 2: Understanding React Components and JSX 17

Introduction 17
What Are Components? 18
  Why Components? • 18
  The Anatomy of a Component • 19
  What Exactly Are Component Functions? • 22
What Does React Do with All These Components? 23
  Built-In Components • 26
  Naming Conventions • 27
JSX vs HTML vs Vanilla JavaScript 28
  Using React without JSX • 30
  JSX Elements Are Treated Like Regular JavaScript Values • 31
  JSX Elements Must Have a Closing Tag • 33

Moving Beyond Static Content 34
Outputting Dynamic Content • 34
Rendering Images • 35
When Should You Split Components? 37
Summary and Key Takeaways 38
What's Next? • 38
Test Your Knowledge! • 39
Apply What You Learned 39
Activity 2.1: Creating a React App to Present Yourself • 39
Activity 2.2: Creating a React App to Log Your Goals for This Book • 41

Chapter 3: Components and Props 43

Introduction 43
Can Components Do More? 43
Using Props in Components 44
Passing Props to Components • 44
Consuming Props in a Component • 45
Components, Props, and Reusability 46
The Special "children" Prop • 46
Which Components Need Props? • 47
How to Deal with Multiple Props • 48
Spreading Props • 49
Prop Chains/Prop Drilling • 51
Summary and Key Takeaways 52
What's Next? • 52
Test Your Knowledge! • 52
Apply What You Learned 52
Activity 3.1: Creating an App to Output Your Goals for This Book • 53

Chapter 4: Working with Events and State 55

Introduction 55
What's the Problem? 56
How Not to Solve the Problem • 56
A Better Incorrect Solution • 58
Improving the Solution by Properly Reacting to Events • 59

Updating State Correctly 62
  A Closer Look at useState() • 63
    A Look Under the Hood of React • 65
Working with Multiple State Values 67
  Using Multiple State Slices • 68
  Managing Merged State Objects • 69
  Updating State Based on Previous State Correctly • 71
  Two-Way Binding • 75
Deriving Values from State 76
  Working with Forms and Form Submission • 79
  Lifting State Up • 81
Summary and Key Takeaways 84
  What's Next? • 84
  Test Your Knowledge! • 85
Apply What You Learned 85
  Activity 4.1: Building a Simple Calculator • 85
  Activity 4.2: Enhancing the Calculator • 86

Chapter 5: Rendering Lists and Conditional Content 89

Introduction 89
What Are Conditional Content and List Data? 90
Rendering Content Conditionally 90
  Different Ways of Rendering Content Conditionally • 94
    Utilizing Ternary Expressions • 94
    Abusing JavaScript Logical Operators • 96
    Get Creative! • 97
    Which Approach is Best? • 98
  Setting Element Tags Conditionally • 98
Outputting List Data 100
  Mapping List Data • 102
  Updating Lists • 104
  A Problem with List Items • 106
    Keys to the Rescue! • 109
Summary and Key Takeaways 110
  What's Next? • 111
  Test Your Knowledge! • 111

Apply What You Learned 111
Activity 5.1: Showing a Conditional Error Message • 112
Activity 5.2: Outputting a List of Products • 113

Chapter 6: Styling React Apps 117

Introduction 117
How Does Styling Work in React Apps? 118
Using Inline Styles • 121
Setting Styles via CSS Classes • 123
Setting Styles Dynamically • 124
Conditional Styles • 126
Combining Multiple Dynamic CSS Classes • 127
Merging Multiple Inline Style Objects • 129
Building Components with Customizable Styles • 129
Customization with Fixed Configuration Options • 130
The Problem with Unscoped Styles 131
Scoped Styles with CSS Modules • 132
The styled-components Library • 135
Use the Tailwind CSS Library for Styling • 137
Using Other CSS or JavaScript Styling Libraries and Frameworks • 140
Summary and Key Takeaways 140
What's Next? • 141
Test Your Knowledge! • 141
Apply What You Learned 141
Activity 6.1: Providing Input Validity Feedback upon Form Submission • 141
Activity 6.2: Using CSS Modules for Style Scoping • 143

Chapter 7: Portals and Refs 145

Introduction 145
A World without Refs 146
Refs versus State 149
Using Refs for More than DOM Access 151
Refs in Custom Components • 154
Controlled versus Uncontrolled Components • 160
React and Where Things End up in the DOM 163
Portals to the Rescue • 165

Summary and Key Takeaways 167
  What's Next? • 168
  Test Your Knowledge! • 168
Apply What You Have Learned 169
  Activity 7.1: Extract User Input Values • 169
  Activity 7.2: Add a Side Drawer • 170

Chapter 8: Handling Side Effects 173

Introduction 173
What's the Problem? 174
Understanding Side Effects 176
  Side Effects Are Not Just about HTTP Requests • 178
Dealing with Side Effects with the useEffect() Hook 179
  How to Use useEffect() • 180
Effects and Their Dependencies 182
  Unnecessary Dependencies • 183
  Cleaning Up after Effects • 185
  Dealing with Multiple Effects • 189
  Functions as Dependencies • 189
  Avoiding Unnecessary Effect Executions • 194
  Effects and Asynchronous Code • 201
  Rules of Hooks • 202
Summary and Key Takeaways 203
  What's Next? • 204
  Test Your Knowledge! • 204
Apply What You Learned 204
  Activity 8.1: Building a Basic Blog • 205

Chapter 9: Handling User Input & Forms with Form Actions 207

Introduction 207
Handling Form Submissions without Actions 208
  Extracting User Input • 208
    Tracking State • 209
    Relying on Refs • 210
    Taking Advantage of the event Object • 211
  Which Solution Is Best? • 213

Handling Form Submissions with Actions 214
Synchronous vs Asynchronous Actions • 215
Behind the Scenes: Actions Are Transitions 217
Managing State Based on Form Submissions 219
Updating UI State with useActionState() • 219
Managing Pending UI State with useActionState() • 222
Handling Pending UI State with useFormStatus() • 224
Performing Optimistic Updates 226
Summary and Key Takeaways 230
What's Next? • 231
Test Your Knowledge! • 231
Apply What You Learned 231
Activity 9.1: Managing a Feedback Form • 231

Chapter 10: Behind the Scenes of React and Optimization Opportunities 235

Introduction 235
Revisiting Component Evaluations and Updates 236
What Happens When a Component Function Is Called • 238
The Virtual DOM vs the Real DOM 239
State Batching • 241
Avoiding Unnecessary Child Component Evaluations • 242
Avoiding Costly Computations • 247
Utilizing useCallback() • 251
Using the React Compiler • 253
Avoiding Unnecessary Code Download 255
Reducing Bundle Sizes via Code Splitting (Lazy Loading) • 255
Strict Mode 261
Debugging Code and the React Developer Tools 262
Summary and Key Takeaways 266
What's Next? • 266
Test Your Knowledge! • 267
Apply What You Learned 267
Activity 10.1: Optimize an Existing App • 267

Chapter 11: Working with Complex State 271

Introduction 271

A Problem with Cross-Component State **272**
Using Context to Handle Multi-Component State **275**
  Providing and Managing Context Values • 276
  Using Context in Nested Components • 281
  Changing Context from Nested Components • 283
Using the Context API Efficiently **284**
  Getting Better Code Completion • 285
  Context or Lifting State Up? • 285
  Outsourcing Context Logic into Separate Components • 286
  Combining Multiple Contexts • 287
Limitations of useState( ) **288**
Managing State with useReducer( ) **291**
  Understanding Reducer Functions • 291
  Dispatching Actions • 293
Summary and Key Takeaways **296**
  What's Next? • 297
  Test Your Knowledge! • 297
Apply What You Learned **297**
  Activity 11.1: Migrating an App to the Context API • 297
  Activity 11.2: Replacing useState() with useReducer() • 299

Chapter 12: Building Custom React Hooks 301

Introduction **301**
Introducing Custom Hooks **301**
  Why Would You Build Custom Hooks? • 303
  A First Custom Hook • 305
Custom Hooks: A Flexible Feature **308**
  Custom Hooks and Parameters • 309
  Custom Hooks and Return Values • 310
A More Complex Example **312**
  Building a First Version of the Custom Hook • 314
  Making the Hook Useful by Returning Values • 316
  Improving Reusability by Accepting an Input Parameter • 317
Using Custom Hooks for Context Access **320**
Summary and Key Takeaways **322**
  What's Next? • 322
  Test Your Knowledge! • 323

Apply What You Learned 323
Activity 12.1: Build a Custom Keyboard Input Hook • 323

Chapter 13: Multipage Apps with React Router 325

Introduction 325
One Page Is Not Enough 326
Getting Started with React Router and Defining Routes 326
Adding Page Navigation • 329
Working with Layouts & Nested Routes • 334
From Link to NavLink • 338
Route Components versus “Normal” Components • 340
From Static to Dynamic Routes 343
Extracting Route Parameters • 345
Creating Dynamic Links • 346
Navigating Programmatically • 348
Redirecting 351
Handling Undefined Routes • 352
Lazy Loading • 352
Summary and Key Takeaways 354
What’s Next? • 355
Test Your Knowledge! • 355
Apply What You Learned 355
Activity 13.1: Creating a Basic Three-Page Website • 355

Chapter 14: Managing Data with React Router 359

Introduction 359
Data Fetching and Routing Are Tightly Coupled 360
Sending HTTP Requests without React Router • 361
Loading Data with React Router 361
Getting Access to Loaded Data • 364
Loading Data for Dynamic Routes • 366
Loaders, Requests, and Client-Side Code • 367
Layouts Revisited 368
Reusing Data across Routes • 372
Handling Errors 374

Onward to Data Submission 376
Working with action() and Form Data • 380
Returning Data Instead of Redirecting • 383
Controlling Which <Form> Triggers Which Action • 385
Reflecting the Current Navigation Status • 386
Submitting Forms Programmatically • 388
Behind-the-Scenes Data Fetching and Submission • 389
Deferring Data Loading • 393
Summary and Key Takeaways 395
What's Next? • 396
Test Your Knowledge! • 396
Apply What You Learned 396
Activity 14.1: A To-Dos App • 396

Chapter 15: Server-side Rendering & Building Fullstack Apps with Next.js 401

Introduction 401
What's the Problem with Client-Side React Apps? 402
Making Sense of Server-side Rendering (SSR) 403
Adding SSR to a React Application 404
Server-side Data Fetching Is Not Trivial • 405
Introducing Next.js 407
Creating Next.js Projects • 408
Working with File-Based Routes • 410
Server-side Rendering with Next.js • 411
Working with Layouts • 412
Managing Internal Navigation • 415
Highlighting Active Links & Using the "use client" Directive • 415
Creating & Using Regular Components • 418
Handling Dynamic Routes • 420
Other Filename Conventions • 424
Diving Deeper into Next.js 424
Summary and Key Takeaways 424
What's Next? • 425
Test Your Knowledge! • 426
Apply What You Learned 426
Activity 15.1: Migrating a Vite-Based React Router App • 426

Chapter 16: React Server Components & Server Actions 429

Introduction 429

The Problem with Server-side Data Fetching 430

Introducing RSCs 430

- Making Sense of RSCs • 431
- Creating & Using RSCs • 433
- Unlocking RSCs in React Projects • 433
- RSCs and Server Actions Can't Be Used in All Projects • 438
- RSCs vs Server-side Rendering • 439
- RSCs vs Client Components • 440
  - *Not All Components Should Be RSCs • 440*
  - *'use client' Affects Child Components, Too! • 442*
  - *Combining RSCs and Client Components • 444*
- Advanced Data Fetching with Next.js • 451
  - *Managing Loading States with Next.js • 451*

From Data Fetching to Data Mutations 453

- Handling Data Mutations with Server Actions • 453
- Unlocking Server Actions in React Projects • 453
- Defining and Triggering Server Actions • 454
- Handling User Input & Updating the UI • 455
- Server Actions and useActionState() • 457
- Storing Server Actions in Separate Files • 460

Summary and Key Takeaways 461

- What's Next? • 462
- Test Your Knowledge! • 463

Apply What You Learned 463

- Activity 16.1: Build a Mini Blog • 463

Chapter 17: Understanding React Suspense & The use() Hook 467

Introduction 467

Showing Granular Fallback Content with Suspense 468

- Using Suspense for Data Fetching with Next.js • 469
- Using Suspense in Other React Projects—Possible, But Tricky • 472
  - *Suspense Does Not Work with useEffect() • 473*
  - *Fetching Data while Rendering—the Incorrect Way • 474*
  - *Getting Suspense Support Is Tricky • 476*

Using Suspense for Data Fetching with Supporting Libraries • 476
use()ing Data while Rendering • 478
Suspense Usage Patterns 483
Revealing Content Together • 484
Revealing Content as Soon as Possible • 485
Nesting Suspended Content • 486
Should You Fetch Data via Suspense or useEffect()? 487
Summary and Key Takeaways 488
What's Next? • 488
Test Your Knowledge! 489
Apply What You Learned 489
Activity 17.1: Implement Suspense in the Mini Blog • 489

Chapter 18: Next Steps and Further Resources 493

Introduction 493
How Should You Proceed? 493
Become a Fullstack React Developer • 494
Interesting Problems to Explore • 494
Build a Shopping Cart • 495
Build an Application's Authentication System (User Signup and Login) • 496
Build an Event Management Website • 496
Common and Popular React Libraries • 497
Using TypeScript • 498
Other Resources • 498
Beyond React for Web Applications • 498
Final Words 499

Other Books You May Enjoy 503

Index 507

Preface

As the most popular JavaScript library for building modern, interactive user interfaces, React is an in-demand framework that'll bring real value to your career or next project. But like any technology, learning React can be tricky, and finding the right teacher can make things a whole lot easier.

Maximilian Schwarzmüller is a bestselling instructor who has helped over three million students worldwide learn how to code, and his latest React video course (*React—The Complete Guide*) has over eight hundred thousand students on Udemy.

Max has written this in-depth reference to help you get to grips with the world of React programming. Simple explanations, relevant examples, and a clear, concise approach make this fast-paced guide the ideal resource for busy developers.

This book distills the core concepts of React and draws together its key features with neat summaries, thus perfectly complementing other in-depth teaching resources. So, whether you've just finished Max's React video course and are looking for a handy reference tool, or you've been using a variety of other learning material and now need a single study guide to bring everything together, this is the ideal companion to support you through your next React projects. Plus, it's fully up to date for React 19, so you can be sure you're ready to go with the latest version.

Who This Book Is For

This book is designed for developers who already have some familiarity with React basics. It can be used as a standalone resource to consolidate understanding or as a companion guide to a more in-depth course. To get the most value from this book, it is recommended that you have some understanding of the fundamentals of JavaScript, HTML, and CSS.

What This Book Covers

Chapter 1, *React – What and Why*, will re-introduce you to React.js. Assuming that React.js is not brand-new to you, this chapter will clarify which problems React solves, which alternatives exist, how React generally works, and how React projects may be created.

Chapter 2, *Understanding React Components and JSX*, will explain the general structure of a React app (a tree of components) and how components are created and used in React apps.

Chapter 3, *Components and Props*, will ensure that you are able to build reusable components by using a key concept called "props".

Chapter 4, *Working with Events and State*, will cover how to work with state in React components, which different options exist (single state vs multiple state slices) and how state changes can be performed and used for UI updates.

Chapter 5, *Rendering Lists and Conditional Content*, will explain how React apps can render lists of content (e.g., lists of user posts) and conditional content (e.g., alert if incorrect values are entered into an input field).

Chapter 6, *Styling React Apps*, will clarify how React components can be styled and how styles can be applied dynamically or conditionally, touching on popular styling solutions like vanilla CSS, Tailwind CSS, styled components, and CSS modules for scoped styles.

Chapter 7, *Portals and Refs*, will explain how direct DOM access and manipulation is facilitated via the "refs" feature that is built-into React. In addition, you will learn how Portals may be used to optimize the rendered DOM element structure.

Chapter 8, *Handling Side Effects*, will discuss the `useEffect` hook, explaining how it works, how it can be configured for different use cases and scenarios, and how side effects can be handled optimally with this React hook.

Chapter 9, *Handling User Input & Forms with Form Actions*, will explore how React simplifies the process of handling forms by allowing you to define client-side form actions that are triggered upon submission.

Chapter 10, *Behind the Scenes of React and Optimization Opportunities*, will take a look behind the scenes of React and dive into core topics like the virtual DOM, state update batching and key optimization techniques that help you avoid unnecessary re-render cycles (and thus improve performance).

Chapter 11, *Working with Complex State*, will explain how the advanced React hook `useReducer` works, when and why you might want to use it and how it, can be used in React components to manage more complex component state with it. In addition, React's Context API will be explored and discussed in depth, allowing you to manage app-wide state with ease.

Chapter 12, Building Custom React Hooks, will build up on the previous chapters and explore how you can build your own, custom React hooks and what the advantage of doing so is.

Chapter 13, Multipage Apps with React Router, will explain what React Router is and how this extra library can be used to build multipage experiences in a React single-page-application.

Chapter 14, Managing Data with React Router, will dive deeper into React Router and explore how this package can also help with fetching and managing data.

Chapter 15, Server-side Rendering & Building Fullstack Apps with Next.js, will help you understand the concept of **server-side rendering** (**SSR**) and help you use your React knowledge with the popular Next.js framework to build applications that span across both the front and backend.

Chapter 16, React Server Components & Server Actions, will build upon the idea of building fullstack React apps and explain how you may render components and handle form submissions on the server side.

Chapter 17, Understanding React Suspense & The use() Hook, will explain how React helps you provide better user experiences by showing fallback content while data is being fetched.

Chapter 18, Next Steps and Further Resources, will cover the core and extended React ecosystem and which resources may be helpful for next steps.

This book also comes with the following downloadable supplementary content:

- A cheatsheet accompanying every chapter of the book
- A video in which author Maximilian gives you his recommendations for next steps after finishing this book
- A video in which author Maximilian shares his thoughts about the future of React

Instructions for claiming this content are available at the end of the *Preface*.

Staying Up to Date with This Book

This edition of this book was written when React 19 was released, though most of the core concepts explained throughout this book have beern around since React 18 or even before that. Thus, the vast majority of the features covered in this book can be considered extremely stable and unlikely to change in the near future.

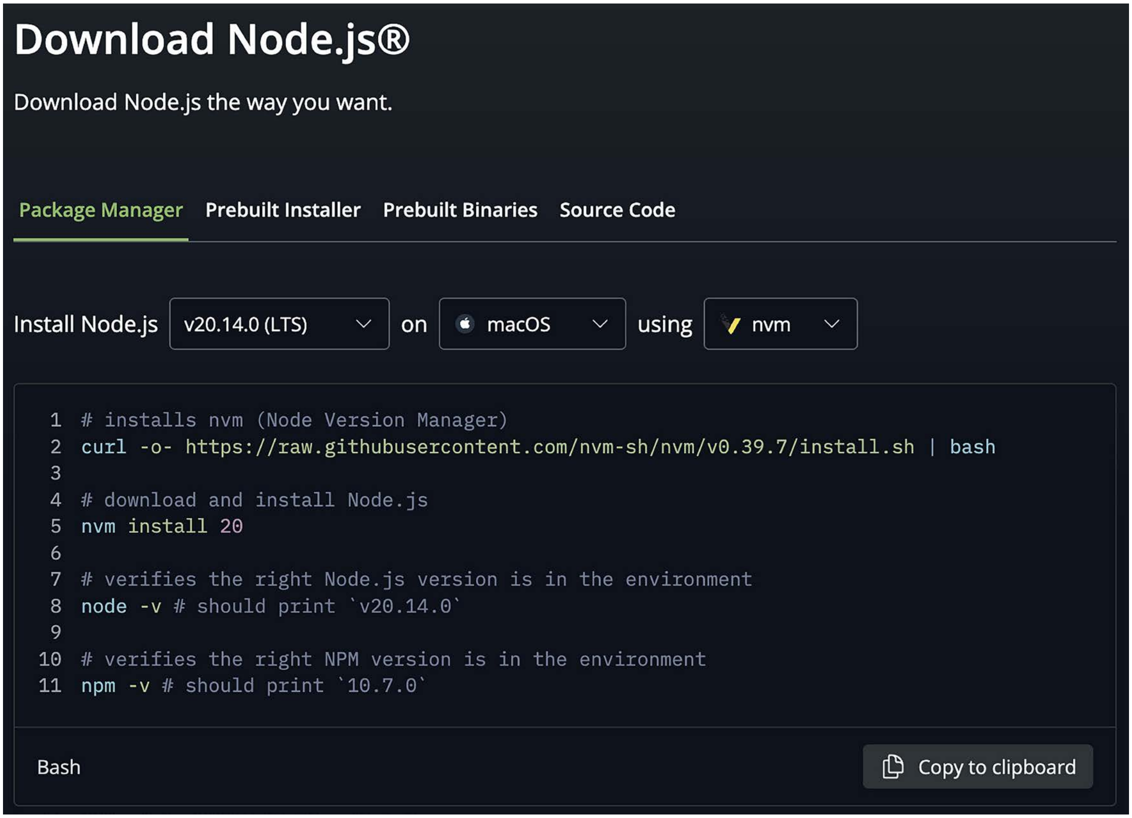
But the book will also cover some relatively new React features, like server components or server actions. Whilst breaking changes are also unlikely for those concepts, a document has been created on GitHub to track any corrections or deviations you should be aware of when reading this book: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/main/CHANGELOG.md`.

Following Along with the Book

Before you can successfully create and run React.js projects on your system, you will need to ensure you have **Node.js** and **npm** (included with your installation by default) installed.

These are available for download at `https://nodejs.org/en/`.

The home page of this site should automatically provide you with the most recent installation options for your platform and system. For more options, select **Downloads** in the site navigation bar. This will open a new page through which you can explore all installation choices for all main platforms, as shown in the screenshot below:



Download Node.js®

Download Node.js the way you want.

Install Node.js v20.14.0 (LTS) on macOS using nvm

```bash
# installs nvm (Node Version Manager)
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash

# download and install Node.js
nvm install 20

# verifies the right Node.js version is in the environment
node -v # should print `v20.14.0`

# verifies the right NPM version is in the environment
npm -v # should print `10.7.0`
```

Installing React.js

React.js projects can be created in various ways, including custom-built project setups that incorporate webpack, babel and other tools. The recommended way for this book is the usage of the Vite tool though. This tool and the process of creating a React app will be covered in *Chapter 1*, *React – What and Why*, but you may refer to this section for step-by-step instructions on this task.

Perform the following steps to create a React.js project on your system:

1. Open your terminal (Powershell/Command Prompt for Windows; bash for Linux).
2. Use the make directory command to create a new project folder with a name of your choosing (e.g., `mkdir react-projects`) and navigate to that directory using the change directory command (e.g., `cd react-projects`).
3. Enter the following command prompt to create a new project directory within this folder:

```
npm create vite@latest my-app
```

After running this command, choose **React** and **JavaScript** when prompted for input.

4. Once completed, navigate to your new directory using the `cd` command:

```
cd my-app
```

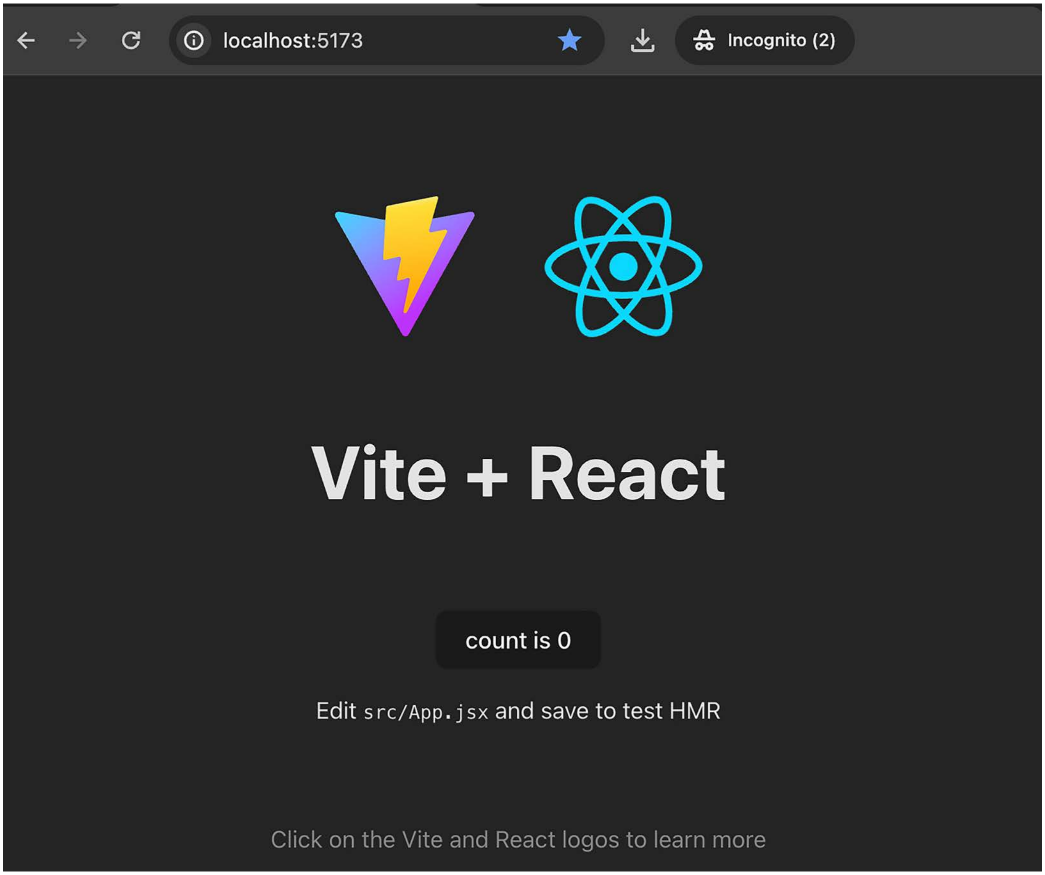
5. Open a terminal window in this new project directory and run the following command to install all required dependencies:

```
npm install
```

6. Once this command is completed, in the same terminal, run the following command to start a Node.js development server:

```
npm run dev
```

7. This command outputs a server address you can visit to preview the React application. By default, the address is `http://localhost:5173`. Type that address in the address/location bar to navigate to `localhost:5173`, as shown in the screenshot below:



localhost:5173
Incognito (2)
Vite + React
count is 0
Edit src/App.jsx and save to test HMR
Click on the Vite and React logos to learn more

8. When you are ready to stop development for the time being, use *Ctrl* + *C* in the same terminal as in *Step 5* to quit running your server. To relaunch it, simply run the `npm run dev` command in that terminal once again. Keep the process started by `npm run dev` up and running while developing, as it will automatically update the website loaded on `localhost:5173` with any changes you make.

Download the Example Code Files

The code bundle for the book is hosted on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the Color Images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781836202271`.

Conventions Used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "Once the root entry point has been defined, a method called `render()` can be called on the `root` object created via `createRoot()`."

A block of code is set as follows:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

import './index.css';
import App from './App.jsx';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
import { memo } from 'react';

import classes from './Error.module.css';

function Error({ message }) {
  console.log('<Error /> component function is executed.');
  if (!message) {
    return null;
  }

  return <p className={classes.error}>{message}</p>;
}

export default memo(Error);
```

Any command-line input or output is written as follows:

```
npm create vite@latest my-react-project
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "React simplifies the creation and management of such UIs by moving from an **imperative** to a **declarative** approach."

> Warnings or important notes appear like this.

> Tips and tricks appear like this.

Get in Touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Share Your Thoughts

Once you've read *React Key Concepts, Second Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download the Free PDF and Supplementary Content

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

Additionally, with this book you get access to supplementary/bonus content for you to learn more. You can use this to add on to your learning journey on top of what you have in the book.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:

   `https://packt.link/supplementary-content-9781836202271`

2. Submit your proof of purchase.
3. Submit your book code. You can find the code on page no. 169 of the book.
4. That's it! We'll send your free PDF, supplementary content, and other benefits to your email directly

Description of Supplementary Content

This book comes with the following bonus material (claimable via the mechanism described above):

- A cheatsheet accompanying every chapter of the book
- A video in which author Maximilian gives you his recommendations for next steps after finishing this book
- A video in which author Maximilian shares his thoughts about the future of React

1

React – What and Why

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Describe what React is and why you would use it
- Compare React to web projects built with just JavaScript
- Explain the difference between imperative and declarative code
- Differentiate between **single-page applications** (**SPAs**) and multi-page apps
- Create new React projects

Introduction

React.js (or just **React**, as it's also called and as it'll be referred to for the majority of this book) is one of the **most popular frontend JavaScript libraries** – maybe even the most popular one, according to a 2023 Stack Overflow developer survey. It is currently used by over 5% of the top 1,000 websites and compared to other popular frontend JavaScript libraries and frameworks like Angular, React is leading by a huge margin, when looking at key metrics like weekly package downloads via **npm**, which is a tool commonly used for downloading and managing JavaScript packages.

Though it is certainly possible to write good React code without fully understanding how React works and why you're using it, you'll likely be able to learn advanced concepts quicker and avoid errors when trying to understand the tools you're working with as well as the reasons for picking a certain tool in the first place.

Therefore, before considering anything about its core concepts and ideas or reviewing example code, you first need to understand what React actually is and why it exists. This will help you understand how React works internally and why it offers the features it does.

If you already know why you're using React, why solutions like React, in general, are being used instead of **vanilla JavaScript** (i.e., JavaScript without any frameworks or libraries, more on this in the next section), and what the idea behind React and its syntax is, you may, of course, skip this section

But if you only *think* that you know it and are not 100% certain, you should definitely read this chapter first.

What is React?

React is a JavaScript library, and if you take a look at the official web page (the official React website and documentation are available at this link: `https://react.dev/`), you learn that the creators call it *"The library for web and native user interfaces."*

But what does this mean?

First, it's important to understand that React is a JavaScript library. As a reader of this book, you know what JavaScript is and why you use JavaScript in the browser. JavaScript allows you to add interactivity to your website since, with JavaScript, you can react to user events and manipulate the page after it is loaded. This is extremely valuable as it allows you to build highly interactive web **user interfaces** (**UIs**).

But what is a "library" and how does React help with building UIs?

While you can have philosophical discussions about what a library is (and how it differs from a framework), the pragmatic definition of a library is that it's a collection of functionalities that you can use in your code to achieve results that would normally require more code and work from your side. Libraries can help you write more concise and possibly also less error-prone code and enable you to implement certain features more quickly.

React is such a library – one that focuses on providing functionalities that help you create interactive and reactive UIs. Indeed, React deals with more than web interfaces (i.e., websites loaded in browsers). You can also build native apps for mobile devices with React and React Native, which is another library that utilizes React under the hood. The React concepts covered in this book still apply, no matter which target platform is chosen. But examples will focus on React for web browsers. No matter which platform you're targeting though, creating interactive UIs with just JavaScript can quickly become very complex and overwhelming.

The Problem with "Vanilla JavaScript"

Vanilla JavaScript is a term commonly used in web development to refer to JavaScript without any frameworks or libraries. That means you write all the JavaScript on your own, without falling back to any libraries or frameworks that would provide extra utility functionalities. When working with vanilla JavaScript, you especially don't use major frontend frameworks or libraries like React or Angular.

Using vanilla JavaScript generally has the advantage that visitors of a website have to download less JavaScript code (as major frameworks and libraries typically are quite sizeable and can quickly add 50+ KB of extra JavaScript code that has to be downloaded).

The downside of relying on vanilla JavaScript is that you, as the developer, must implement all functionalities from the ground up on your own. This can be error prone and highly time consuming. Therefore, especially more complex UIs and websites can quickly become very hard to manage with vanilla JavaScript.

React simplifies the creation and management of such UIs by moving from an **imperative** to a **declarative** approach. Though this is a nice sentence, it can be hard to grasp if you haven't worked with React or similar frameworks before. To understand it, the idea behind "imperative versus declarative approaches," and why you might want to use React instead of just vanilla JavaScript, it's helpful to take a step back and evaluate how vanilla JavaScript works.

Let's look at a short code snippet that shows how you could handle the following UI actions with vanilla JavaScript:

1. Add an event listener to a button to listen for `click` events.
2. Replace the text of a paragraph with new text once a click on the button occurs.

```
const buttonElement = document.querySelector('button');
const paragraphElement = document.querySelector('p');

function updateTextHandler() {
  paragraphElement.textContent = 'Text was changed!';
}

buttonElement.addEventListener('click', updateTextHandler);
```

This example is deliberately kept simple, so it's probably not looking too bad or overwhelming. It's just a basic example to show how code is generally written with vanilla JavaScript (a more complex example will be discussed later). But even though this example is straightforward to digest, working with vanilla JavaScript will quickly reach its limits for feature-rich UIs and the code to handle various user interactions accordingly also becomes more complex. Code can quickly grow significantly, so maintaining it can become a challenge.

In the preceding example, code is written with vanilla JavaScript and, as a consequence, imperatively. This means that you write instruction after instruction, and you describe every step that needs to be taken in detail.

The code shown previously could be translated into these more human-readable instructions:

1. Look for an **HTML** element of the `button` type to obtain a reference to the first button on the page.
2. Create a constant (i.e., a data container) named `buttonElement` that holds that button reference.
3. Repeat *Step 1* but get a reference to the first element that is of type of `p`.
4. Store the paragraph element reference in a constant named `paragraphElement`.
5. Add an event listener to the `buttonElement` that listens for `click` events and triggers the `updateTextHandler` function whenever such a `click` event occurs.
6. Inside the `updateTextHandler` function, use the `paragraphElement` to set its `textContent` to `"Text was changed!"`.

Do you see how every step that needs to be taken is clearly defined and written out in the code?

This shouldn't be too surprising because that is how most programming languages work: you define a series of steps that must be executed in order. It's an approach that makes a lot of sense because the order of code execution shouldn't be random or unpredictable.

However, when working with UIs, this imperative approach is not ideal. Indeed, it can quickly become cumbersome because, as a developer, you have to add a lot of instructions that, despite adding little value, cannot simply be omitted. You need to write all the **Document Object Model** (**DOM**) instructions that allow your code to interact with elements, add elements, manipulate elements, and so on.

Your core business logic (e.g., deriving and defining the actual text that should be set after a click) therefore often makes up only a small chunk of the overall code. When controlling and manipulating web UIs with JavaScript, a huge chunk (often the majority) of your code is frequently made up of DOM instructions, event listeners, HTML element operations, and UI state management.

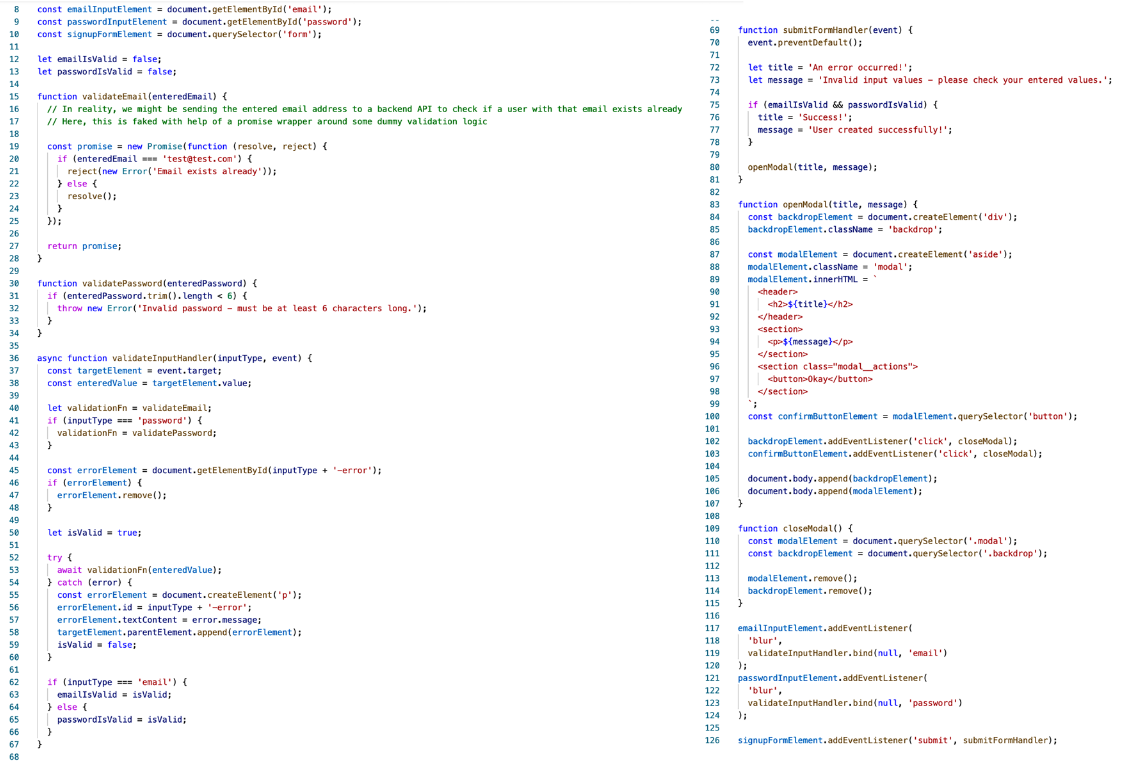
As a result, you end up describing all the steps that are required to interact with the UI technically **and** all the steps that are required to derive the output data (i.e., the desired final state of the UI).

> **Note**
>
> This book assumes that you are familiar with the DOM. In a nutshell, the DOM is the "bridge" between your JavaScript code and the HTML code of the website with which you want to interact. Via the built-in **DOM API**, JavaScript is able to create, insert, manipulate, delete, and read HTML elements and their content.
>
> You can learn more about the DOM in this article: `https://academind.com/tutorials/what-is-the-dom`.
>
> Modern web UIs are often quite complex, with lots of interactivity going on behind the scenes. Your website might need to listen for user input in an input field, send that entered data to a server to validate it, output a validation feedback message on the screen, and show an error overlay modal if incorrect data is submitted.

The button-clicking example is not a complex example in general, but the vanilla JavaScript code for implementing such a scenario can be overwhelming. You end up with lots of DOM selection, insertion, and manipulation operations, as well as multiple lines of code that do nothing but manage event listeners. Also, keeping the DOM updated, without introducing bugs or errors, can be a nightmare since you must ensure that you update the right DOM element with the right value at the right time. Here, you will find a screenshot of some example code for the described use case.

> **Note**
>
> The full, working, code can be found on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/01-what-is-react/examples/example-1/vanilla-javascript`.

If you take a look at the JavaScript code in the screenshot (or in the linked repository), you will probably be able to imagine how a more complex UI is likely to look.



```javascript
const emailInputElement = document.getElementById('email');
const passwordInputElement = document.getElementById('password');
const signupFormElement = document.querySelector('form');

let emailIsValid = false;
let passwordIsValid = false;

function validateEmail(enteredEmail) {
  // In reality, we might be sending the entered email address to a backend API to check if a user with that email exists already
  // Here, this is faked with help of a promise wrapper around some dummy validation logic

  const promise = new Promise(function (resolve, reject) {
    if (enteredEmail === 'test@test.com') {
      reject(new Error('Email exists already'));
    } else {
      resolve();
    }
  });

  return promise;
}

function validatePassword(enteredPassword) {
  if (enteredPassword.trim().length < 6) {
    throw new Error('Invalid password - must be at least 6 characters long.');
  }
}

async function validateInputHandler(inputType, event) {
  const targetElement = event.target;
  const enteredValue = targetElement.value;

  let validationFn = validateEmail;
  if (inputType === 'password') {
    validationFn = validatePassword;
  }

  const errorElement = document.getElementById(inputType + '-error');
  if (errorElement) {
    errorElement.remove();
  }

  let isValid = true;

  try {
    await validationFn(enteredValue);
  } catch (error) {
    const errorElement = document.createElement('p');
    errorElement.id = inputType + '-error';
    errorElement.textContent = error.message;
    targetElement.parentElement.append(errorElement);
    isValid = false;
  }

  if (inputType === 'email') {
    emailIsValid = isValid;
  } else {
    passwordIsValid = isValid;
  }
}

function submitFormHandler(event) {
  event.preventDefault();

  let title = 'An error occurred!';
  let message = 'Invalid input values - please check your entered values.';

  if (emailIsValid && passwordIsValid) {
    title = 'Success!';
    message = 'User created successfully!';
  }

  openModal(title, message);
}

function openModal(title, message) {
  const backdropElement = document.createElement('div');
  backdropElement.className = 'backdrop';

  const modalElement = document.createElement('aside');
  modalElement.className = 'modal';
  modalElement.innerHTML = `
    <header>
      <h2>${title}</h2>
    </header>
    <section>
      <p>${message}</p>
    </section>
    <section class="modal__actions">
      <button>Okay</button>
    </section>
  `;
  const confirmButtonElement = modalElement.querySelector('button');

  backdropElement.addEventListener('click', closeModal);
  confirmButtonElement.addEventListener('click', closeModal);

  document.body.append(backdropElement);
  document.body.append(modalElement);
}

function closeModal() {
  const modalElement = document.querySelector('.modal');
  const backdropElement = document.querySelector('.backdrop');

  modalElement.remove();
  backdropElement.remove();
}

emailInputElement.addEventListener(
  'blur',
  validateInputHandler.bind(null, 'email')
);
passwordInputElement.addEventListener(
  'blur',
  validateInputHandler.bind(null, 'password')
);

signupFormElement.addEventListener('submit', submitFormHandler);
```

Figure 1.1: An example JavaScript code file that contains over 100 lines of code for a fairly trivial UI

This example JavaScript file already contains roughly 110 lines of code. Even after minifying ("minifying" means that code is shortened automatically, e.g., by replacing long variable names with shorter ones and removing redundant whitespace; in this case, via `https://www.toptal.com/developers/javascript-minifier`) it and splitting the code across multiple lines thereafter (to count the raw lines of code), it still has around 80 lines of code. That's a full 80 lines of code for a simple UI with only basic functionality. The actual business logic (i.e., input validation, determining whether and when overlays should be shown, and defining the output text) only makes up a small fraction of the overall code base – around 20 to 30 lines of code, in this case (around 20 after minifying).

That's roughly 75% of the code spent on pure DOM interaction, DOM state management, and similar boilerplate tasks.

As you can see by these examples and numbers, controlling all the UI elements and their different states (e.g., whether an info box is visible or not) is a challenging task, and trying to create such interfaces with just JavaScript often leads to complex code that might even contain errors.

That's why the imperative approach, wherein you must define and write down every single step, has its limits in situations like this. This is the reason why React provides utility functionalities that allow you to write code differently: with a declarative approach.

Note

This is not a scientific paper, and the preceding example is not meant to act as an exact scientific study. Depending on how you count lines and which kind of code you consider to be "core business logic," you will end up with higher or lower percentage values. The key message doesn't change though: lots of code (in this case most of it) deals with the DOM and DOM manipulation – not with the actual logic that defines your website and its key features.

React and Declarative Code

Coming back to the first, simple code snippet from earlier, here's that same code snippet, this time using React:

```
import { useState } from 'react';

function App() {
  const [outputText, setOutputText] = useState('Initial text');

  function updateTextHandler() {
    setOutputText('Text was changed!');
  }

  return (
    <>
      <button onClick={updateTextHandler}>
        Click to change text
      </button>
      <p>{outputText}</p>
    </>
  );
}
```

This snippet performs the same operations as the first did with just vanilla JavaScript:

1. Add an event listener to a button to listen for `click` events (now with some React-specific syntax: `onClick={…}`).
2. Replace the text of a paragraph with a new text once the click on the button occurs.

Nonetheless, this code looks totally different – like a mixture of JavaScript and HTML. Indeed, React uses a syntax extension called **JSX** (i.e., JavaScript extended to include XML-like syntax). For the moment, it's enough to understand that this JSX code will work because of a **pre-processing** (or **transpilation**) step that's part of the build workflow of every React project.

Pre-processing means that certain tools, which are part of React projects, analyze and transform the code before it is deployed. This allows for development-only syntax like JSX, which would not work in the browser and is for that reason transformed to regular JavaScript before deployment. (You'll get a thorough introduction to JSX in *Chapter 2*, *Understanding React Components and JSX*.)

In addition, the snippet shown previously contains a React-specific feature: State. state will be discussed in greater detail later in the book (*Chapter 4*, *Working with Events and State*, will focus on handling events and states with React). For the moment, you can think of this state as a variable that, when changed, will trigger React to update the UI in the browser.

What you see in the preceding example is the "declarative approach" used by React: you write your JavaScript logic (e.g., functions that should eventually be executed), and you combine that logic with the HTML code that should trigger it or that is affected by it. You don't write the instructions for selecting certain DOM elements or changing the text content of some DOM elements. Instead, with React and JSX, you focus on your JavaScript business logic and define the desired HTML output that should eventually be reached. This output can, and typically will, contain dynamic values that are derived inside of your main JavaScript code.

In the preceding example, `outputText` is some state managed by React. In the code, the `updateTextHandler` function is triggered upon a click, and the `outputText` state value is set to a new string value (`'Text was changed!'`) with the help of the `setOutputText` function. The exact details of what's going on here will be explored in *Chapter 4*.

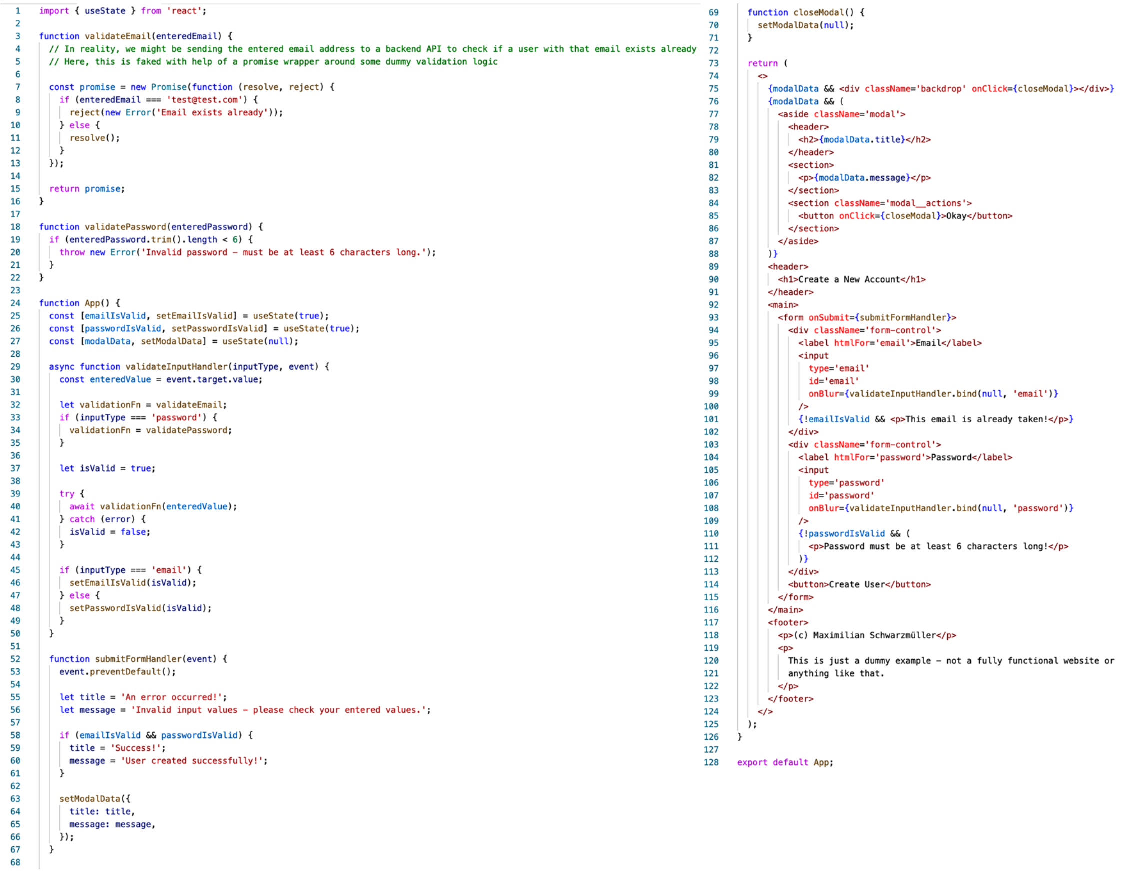
The general idea, though, is that the state value is changed and, since it's being referenced in the last paragraph (`<p>{outputText}</p>`), React outputs the current state value in that place in the actual DOM (and hence, on the actual web page). React will keep the paragraph updated, and therefore, whenever `outputText` changes, React will select this paragraph element again and update its `textContent` automatically.

This is the declarative approach in action. As a developer, you don't need to worry about the technical details (for example, selecting the paragraph and updating its `textContent`). Instead, you will hand this work off to React. You will only need to focus on the desired end states where the goal simply is to output the current value of `outputText` in a specific place (i.e., in the second paragraph in this case) on the page. It's React's job to do the "*behind the scenes*" work of getting to that result.

It turns out that this code snippet isn't shorter than the vanilla JavaScript one; indeed, it's actually even a bit longer. But that's only the case because this first snippet was deliberately kept simple and concise. In such cases, React actually adds a bit of overhead code. If that were your entire UI, using React indeed wouldn't make too much sense. Again, this snippet was chosen because it allows us to see the differences at a glance. Things change if you take a look at the more complex vanilla JavaScript example from before and compare that to its React alternative.

Note

Referenced code can be found on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/01-what-is-react/examples/example-1/vanilla-javascript` and `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/01-what-is-react/examples/example-1/reactjs`, respectively.



```
import { useState } from 'react';

function validateEmail(enteredEmail) {
  // In reality, we might be sending the entered email address to a backend API to check if a user with that email exists already
  // Here, this is faked with help of a promise wrapper around some dummy validation logic

  const promise = new Promise(function (resolve, reject) {
    if (enteredEmail === 'test@test.com') {
      reject(new Error('Email exists already'));
    } else {
      resolve();
    }
  });

  return promise;
}

function validatePassword(enteredPassword) {
  if (enteredPassword.trim().length < 6) {
    throw new Error('Invalid password - must be at least 6 characters long.');
  }
}

function App() {
  const [emailIsValid, setEmailIsValid] = useState(true);
  const [passwordIsValid, setPasswordIsValid] = useState(true);
  const [modalData, setModalData] = useState(null);

  async function validateInputHandler(inputType, event) {
    const enteredValue = event.target.value;

    let validationFn = validateEmail;
    if (inputType === 'password') {
      validationFn = validatePassword;
    }

    let isValid = true;

    try {
      await validationFn(enteredValue);
    } catch (error) {
      isValid = false;
    }

    if (inputType === 'email') {
      setEmailIsValid(isValid);
    } else {
      setPasswordIsValid(isValid);
    }
  }

  function submitFormHandler(event) {
    event.preventDefault();

    let title = 'An error occurred!';
    let message = 'Invalid input values - please check your entered values.';

    if (emailIsValid && passwordIsValid) {
      title = 'Success!';
      message = 'User created successfully!';
    }

    setModalData({
      title: title,
      message: message,
    });
  }

  function closeModal() {
    setModalData(null);
  }

  return (
    <>
      {modalData && <div className='backdrop' onClick={closeModal}></div>}
      {modalData && (
        <aside className='modal'>
          <header>
            <h2>{modalData.title}</h2>
          </header>
          <section>
            <p>{modalData.message}</p>
          </section>
          <section className='modal__actions'>
            <button onClick={closeModal}>Okay</button>
          </section>
        </aside>
      )}
      <header>
        <h1>Create a New Account</h1>
      </header>
      <main>
        <form onSubmit={submitFormHandler}>
          <div className='form-control'>
            <label htmlFor='email'>Email</label>
            <input
              type='email'
              id='email'
              onBlur={validateInputHandler.bind(null, 'email')}
            />
            {!emailIsValid && <p>This email is already taken!</p>}
          </div>
          <div className='form-control'>
            <label htmlFor='password'>Password</label>
            <input
              type='password'
              id='password'
              onBlur={validateInputHandler.bind(null, 'password')}
            />
            {!passwordIsValid && (
              <p>Password must be at least 6 characters long!</p>
            )}
          </div>
          <button>Create User</button>
        </form>
      </main>
      <footer>
        <p>(c) Maximilian Schwarzmüller</p>
        <p>
          This is just a dummy example - not a fully functional website or
          anything like that.
        </p>
      </footer>
    </>
  );
}

export default App;
```

Figure 1.2: The code snippet from before is now implemented via React

It's still not short because all the JSX code (i.e., the HTML output) is included in the JavaScript file. If you ignore pretty much the entire right side of that screenshot (since HTML was not part of the vanilla JavaScript files either), the React code gets much more concise. However, most importantly, if you take a closer look at all the React code (also in the first, shorter snippet), you will notice that there are absolutely no operations that would select DOM elements, create or insert DOM elements, or edit DOM elements.

This is the core idea of React. You don't write down all the individual steps and instructions; instead, you focus on the "big picture" and the desired end states of your page content. With React, you can merge your JavaScript and markup code without having to deal with the low-level instructions of interacting with the DOM like selecting elements via `document.getElementById()` or similar operations.

Using this declarative approach instead of the imperative approach with vanilla JavaScript allows you, the developer, to focus on your core business logic and the different states of your HTML code. You don't need to define all the individual steps that have to be taken (like "adding an event listener," "selecting a paragraph," etc.), and this simplifies the development of complex UIs tremendously.

Note

It is worth emphasizing that React is not a great solution if you're working on a very simple UI. If you can solve a problem with a few lines of vanilla JavaScript code, there is probably no strong reason to integrate React into the project.

Looking at React code for the first time, it can look very unfamiliar and strange. It's not what you're used to from JavaScript. Still, it is JavaScript – just enhanced with this JSX feature and various React-specific functionalities (like state). It may be less confusing if you remember that you typically define your UI (i.e., your content and its structure) with HTML. You don't write step-by-step instructions there either but rather create a nested tree structure with HTML tags. You express your content, the meaning of different elements, and the hierarchy of your UI by using different HTML elements and nesting HTML tags.

If you keep this in mind, the "traditional" (vanilla JavaScript) approach of manipulating the UI should seem rather odd. Why would you start defining low-level instructions like *"insert a paragraph element below this button and set its text to <some text>"* if you don't do that in HTML at all? React, in the end, brings back that HTML syntax, which is far more convenient when it comes to defining content and structure. With React, you can write dynamic JavaScript code side by side with the UI code (i.e., the HTML code) that is affected by it or related to it.

How React Manipulates the DOM

As mentioned earlier, when writing React code, you typically write it as shown previously: you blend HTML with JavaScript code by using the JSX syntax extension.

It is worth pointing out that JSX code does not run like this in browsers. It instead needs to be pre-processed before deployment. The JSX code must be transformed into regular JavaScript code before being served to browsers. The next chapter will take a closer look at JSX and what it's transformed into. For the moment, though, simply keep in mind that JSX code must be transformed.

Nonetheless, it is worth knowing that the code to which JSX will be transformed will also not contain any DOM instructions. Instead, the transformed code will execute various utility methods and functions that are built into React (in other words, those that are provided by the React package that need to be added to every React project). Internally, React creates a virtual DOM-like tree structure that reflects the current state of the UI. This book takes a closer look at this abstract, virtual DOM, and how React works in *Chapter 10*, *Behind the Scenes of React and Optimization Opportunities*. That's why React (the library) splits its core logic across two main packages:

- The main `react` package
- The `react-dom` package

The main `react` package is a third-party JavaScript library that needs to be imported into a project to use React's features (like JSX or state) there. It's this package that creates this virtual DOM and derives the current UI state. But you also need the `react-dom` package in your project if you want to manipulate the DOM with React.

The `react-dom` package, specifically the `react-dom/client` part of that package, acts as a "translation bridge" between your React code, the internally generated virtual DOM, and the browser with its actual DOM that needs to be updated. It's the `react-dom` package that will produce the actual DOM instructions that will select, update, delete, and create DOM elements.

This split exists because you can also use React with other target environments. A very popular and well-known alternative to the DOM (i.e., to the browser) would be React Native, which allows developers to build native mobile apps with the help of React. With React Native, you also include the react package in your project, but in place of `react-dom`, you would use the `react-native` package. In this book, "React" refers to both the `react` package and the "bridge" packages (like `react-dom`).

Note

As mentioned earlier, this book focuses on React itself. The concepts explained in this book, therefore, will apply to both web browsers and websites as well as mobile devices. Nonetheless, all examples will focus on the web and `react-dom` since that avoids introducing extra complexity.

Introducing SPAs

React can be used to simplify the creation of complex UIs, and there are two main ways of doing that:

- Manage parts of a website (e.g., a chat box in the bottom left corner).
- Manage the entire page and all user interactions that occur on it.

Both approaches are viable, but the more popular and common scenario is the second one: using React to manage the entire web page, instead of just parts of it. This approach is more popular because most websites that have complex UIs have not just one, but multiple complex elements on their pages. Complexity would actually increase if you were to start using React for some website parts without using it for other areas of the site. For this reason, it's very common to manage the entire website with React.

This doesn't even stop after using React on one specific page of the site. Indeed, React can be used to handle URL path changes and update the parts of the page that need to be updated in order to reflect the new page that should be loaded. This functionality is called **routing** and third-party packages like `react-router-dom` (see *Chapter 13*, *Multipage Apps with React Router*), which integrate with React, allow you to create a website wherein the entire UI is controlled via React.

A website that does not just use React for parts of its pages but instead for all subpages and for routing is often built as a SPA because it's common to create React projects that contain only one HTML file (typically named `index.html`), which is used to initially load the React JavaScript code. Thereafter, the React library and your React code take over and control the actual UI. This means that the entire UI is created and managed by JavaScript via React and your React code.

That being said, it's also becoming more and more popular to build full-stack React apps, where frontend and backend code are merged. Modern React frameworks like **Next.js** simplify the process of building such web apps. Whilst the core concepts are the same, no matter which kind of application is built, this book will explore full-stack React app development in greater detail in *Chapter 15*, *Server-side Rendering & Building Fullstack Apps with Next.js*, *Chapter 16*, *React Server Components and Server Actions* and *Chapter 17*, *Understanding React Suspense and the use() Hook*.

Ultimately, this book prepares you for working with React on all kinds of React projects since the core building blocks and key concepts are always the same.

Creating a React Project with Vite

To work with React, the first step is the creation of a React project. The official documentation recommends using a framework like Next.js. But while this might make sense for complex web applications, it's overwhelming for getting started with React and for exploring React concepts. Next.js and other frameworks introduce their own concepts and syntax. As a result, learning React can quickly become frustrating since it can be difficult to tell React features apart from framework features. In addition, not all React apps need to be built as full-stack web apps – consequently, using a framework like Next.js might add unnecessary complexity.

That's why Vite-based React projects have emerged as a popular alternative. **Vite** is an open-source development and build tool that can be used to create and run web development projects based on all kinds of libraries and frameworks – React is just one of the many options.

Vite creates projects that come with a built-in, preconfigured build process that, in the case of React projects, takes care of the JSX code transpilation. It also provides a development web server that runs locally on your system and allows you to preview the React app while you're working on it.

You need a project setup like this because React projects typically use features like JSX, which wouldn't work in the browser without prior code transformation. Hence, as mentioned earlier, a pre-processing step is required.

To create a project with Vite, you must have Node.js installed – preferably the latest (or latest **LTS**) version. You can get the official Node.js installer for all operating systems from `https://nodejs.org/`. Once you have installed Node.js, you will also gain access to the built-in `npm` command, which you can use to utilize the Vite package to create a new React project.

You can run the following command inside of your command prompt (Windows), bash (Linux), or terminal (macOS) program. Just make sure that you navigate (via `cd`) into the folder in which you want to create your new project:

```
npm create vite@latest my-react-project
```

Once executed, this command will prompt you to choose a framework or library you want to use for this new project. You should choose React and then JavaScript.

This command will create a new subfolder with a basic React project setup (i.e., with various files and folders) in the place where you ran it. You should run it in some path on your system where you have full read and write access and where you're not conflicting with any system or other project files.

It's worth noting that the project creation command does not install any required dependencies such as the React library packages. For that reason, you must navigate into the created folder in your system terminal or command prompt (via `cd my-react-project`) and install these packages by running the following command:

```
npm install
```

Once the installation finishes successfully, the project setup process is complete.

To view the created React application, you can start a development server on your machine via this command:

```
npm run dev
```

This invokes a script provided by Vite, which will spin up a locally running web server that pre-processes, builds, and hosts your React-powered SPA – by default on `localhost:5173`. Therefore, while working on the code, you typically have this development server up and running as it allows you to preview and test code changes.

Best of all, this local development server will automatically update the website whenever you save any code changes, hence allowing you to preview your changes almost instantly.

You can quit this server whenever you're done for the day by pressing *Ctrl* + *C* in the terminal or command prompt where you executed `npm run dev`.

Whenever you're ready to start working on the project again, you can restart the server via `npm run dev`.

> **Note**
>
> In case you encounter any issues with creating a React project, you can also download and use the following starting project: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/01-what-is-react/react-starting-project`. It's a project created via Vite, which can be used in the same way as if it were created with the preceding command.
>
> When using this starting project (or, in fact, any GitHub-hosted code snapshot belonging to this book), you need to run `npm install` in the project folder first, before running `npm run dev`.

The exact project structure (that is, the file names and folder names) may vary over time, but generally, every new Vite-based React project contains a couple of key files and folders:

- A `src/` folder, which contains the main source code files for the project:
    - A `main.jsx` file, which is the main entry script file that will be executed first

    - An `App.jsx` file, which contains the root component of the application (you'll learn more about components in the next chapter)
    - Various styling (`*.css`) files, which are imported by the JavaScript files
    - An `assets/` folder that can be used to store images or other assets that should be used in your React code
- A `public/` folder, which contains static files that will be part of the final website (e.g., a favicon)
- An `index.html` file, which is the single HTML page of this website
- `package.json` and `package-lock.json` are files that list and define the third-party dependencies of your project:
    - Production dependencies like `react` or `react-dom`
    - Development dependencies like `eslint` for automated code quality checks
- Other project configuration files (e.g., `.gitignore` for managing Git file tracking)
- A `node_modules` folder, which contains the actual code of the installed third-party packages

It's worth noting that `App.jsx` and `main.jsx` use `.jsx` as a file extension, not `.js`. This is a file extension that's enforced by Vite for files that do not just contain standard JavaScript but also JSX code. When working on a Vite project, most of your project files will consequently use `.jsx` as an extension.

Almost all of the React-specific code will be written in the `App.jsx` file or custom component files that will be added to the project. We will explore components in the next chapter.

Note

`package.json` is the file in which you actually manage packages and their versions. `package-lock.json` is created automatically (by `Node.js`). It locks in exact dependency and sub-dependency versions, whereas `package.json` only specifies version ranges. You can learn more about these files and package versions at `https://docs.npmjs.com/`.

The code of the project's dependencies is stored in the `node_modules` folder. This folder can become very big since it contains the code of all installed packages and their dependencies. For that reason, it's typically not included if projects are shared with other developers or pushed to GitHub. The `package.json` file is all you need. By running `npm install`, the `node_modules` folder will be recreated locally.

Summary and Key Takeaways

- React is a library, though it's actually a combination of two main packages: `react` and `react-dom`.
- Though it is possible to build non-trivial UIs without React, simply using vanilla JavaScript to do so can be cumbersome, error prone, and hard to maintain.
- React simplifies the creation of complex UIs by providing a declarative way to define the desired end states of the UI.

- **Declarative** means that you define the target UI content and structure, combined with different states (e.g., *"Is a modal open or closed?"*), and you leave it up to React to figure out the appropriate DOM instructions.
- The react package itself derives UI states and manages a virtual DOM. It is a "bridge," like `react-dom` or `react-native`, that translates this virtual DOM into actual UI (DOM) instructions.
- With React, you can build SPAs, meaning that React is used to control the entire UI on all pages as well as the routing between pages.
- You can also use React, in combination with frameworks like Next.js, to build full-stack web applications where server- and client-side code are connected.
- React projects can be created with the help of the Vite package, which provides a readily configured project folder and a live preview development server.

What's Next?

At this point, you should have a basic understanding of what React is and why you might consider using it, especially for building non-trivial UIs. You learned how to create new React projects with Vite, and you are now ready to dive deeper into React and the actual key features it offers.

In the next chapter, you will learn about a concept called **components**, which are the fundamental building blocks of React apps. You will learn how components are used to compose UIs and why those components are needed in the first place. The next chapter will also dive deeper into JSX and explore how it is transformed into regular JavaScript code and which kind of code you could write alternatively to JSX.

Test Your Knowledge!

Test your knowledge about the concepts covered in this chapter by answering the following questions. You can then compare your answers to example answers that can be found here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/01-what-is-react/exercises/questions-answers.md`.

1. What is React?
2. Which advantage does React offer over vanilla JavaScript projects?
3. What's the difference between imperative and declarative code?
4. What is a **Single-Page-Application** (**SPA**)?
5. How can you create new React projects and why do you need such a complex project setup?

Join Us on Discord

Read this book alongside other users, AI experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the author via Ask Me Anything sessions, and much more.

Scan the QR code or visit the link to join the community.

`https://packt.link/ReactKeyConcepts2e`

2

Understanding React Components and JSX

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Define what exactly components are
- Build and use components effectively
- Utilize common naming conventions and code patterns
- Describe the relationship between components and JSX
- Write JSX code and understand why it's used
- Write React components without using JSX code
- Write your first React apps

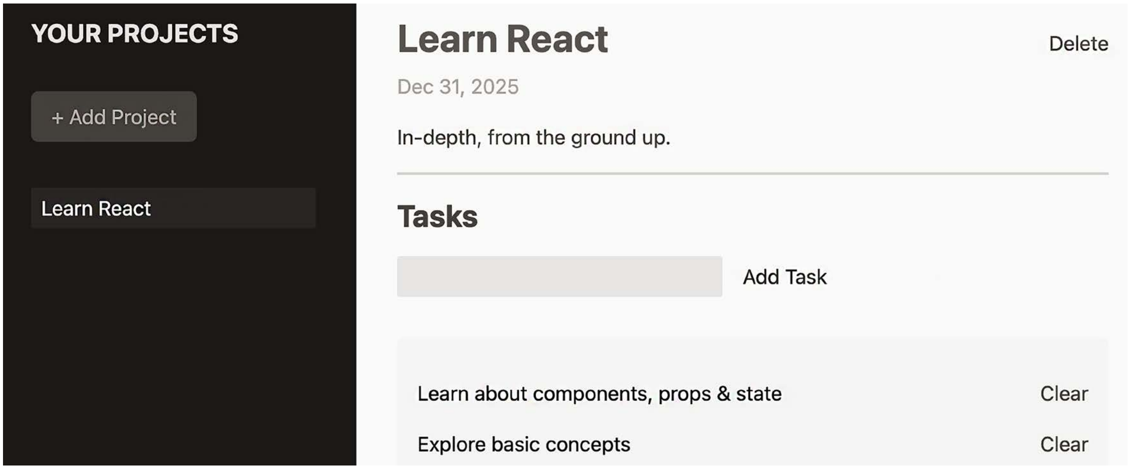
Introduction

In the previous chapter, you learned about React in general, what it is, and why you should consider using it for building user interfaces. You also learned how to create React projects with the help of Vite, by running `npm create vite@latest <your-project-name>`.

In this chapter, you will learn about one of the most important React concepts and building blocks. You will learn that components are reusable building blocks that are used to build user interfaces. In addition, JSX code will be discussed in greater detail so that you will be able to use the concept of components and JSX to build your own first basic React apps.

What Are Components?

A key concept of React is the usage of so-called components. **Components** are reusable building blocks that are combined to compose the final user interface. For example, a basic website could be made up of a sidebar that includes navigation items and a main section that includes elements for adding and viewing tasks.



In-depth, from the ground up.

Learn about components, props & state

Explore basic concepts

Figure 2.1: An example task management screen with sidebar and main area

If you look at this example page, you might be able to identify various building blocks (i.e., components). Some of these components are even reused:

- The sidebar and its navigation items
- The main page area
- In the main area, the header with the title and due date
- A form for adding tasks
- A list of tasks

Please note that some components are nested inside other components—i.e., components are also made up of other components. That's a key feature of React and similar libraries.

Why Components?

No matter which web page you look at, they are all made up of building blocks like this. It's not a React-specific concept or idea. Indeed, HTML itself "thinks" in components if you take a closer look. You have elements like `<img>`, `<header>`, `<nav>`, etc., and you combine these elements to describe and structure your website content.

But React **embraces** this idea of breaking a web page into reusable building blocks because it is an approach that allows developers to work on small, manageable chunks of code. It's easier and more maintainable than working on a single, huge HTML (or React code) file.

That's why other libraries—both frontend libraries like React or Angular as well as backend libraries and templating engines like **EJS** (**Embedded JavaScript templates**)—also embrace components (though the names might differ, you also find *"partials"* or *"includes"* as common names).

Note

EJS is a popular templating engine for JavaScript. It's especially popular for backend web development with Node.js.

When working with React, it's especially important to keep your code manageable and work with small, reusable components because React components are not just collections of HTML code. Instead, a React component also encapsulates JavaScript logic and often also CSS styling. For complex user interfaces, the combination of markup (JSX), logic (JavaScript), and styling (CSS) could quickly lead to large chunks of code, thus making it difficult to maintain that code. Think of a large HTML file that also includes JavaScript and CSS code. Working in such a code file wouldn't be a lot of fun.

To make a long story short, when working on a React project, you will work with lots of components. You will split your code into small, manageable building blocks and then combine these components to form the overall user interface. It's a key feature of React.

Note

When working with React, you should embrace this idea of working with components. But technically, they're optional. You could, theoretically, build very complex web pages with one single component alone. It would not be much fun, and it would not be practical, but it would technically be possible without any issues.

The Anatomy of a Component

Components are important. But what exactly does a React component look like? How do you write React components on your own?

Here's an example component:

```
import { useState } from 'react';

function SubmitButton() {
  const [isSubmitted, setIsSubmitted] = useState(false);

  function handleSubmit() {
    setIsSubmitted(true);
  };

  return (
      <button onClick={handleSubmit}>
        { isSubmitted ? 'Loading…' : 'Submit' }
      </button>
    );
  };

  export default SubmitButton;
```

Typically, you would store a code snippet like this in a separate file (e.g., a file named `SubmitButton.jsx`, stored inside a `/components` folder, which in turn resides in the `/src` folder of your React project) and import it into other component files that need this component. `.jsx` is used as an extension since the file contains JSX code. Vite enforces the usage of `.jsx` as a file extension if you're writing JSX code – storing such code in `.js` files is not allowed in Vite projects (even though it might work in other React project setups).

The following component imports the component defined above and uses it in its `return` statement to output the `SubmitButton` component:

```
import SubmitButton from './submit-button.jsx';

function AuthForm() {
  return (
    <form>
      <input type="text" />
      <SubmitButton />
    </form>
  );
};

export default AuthForm;
```

The `import` statements you see in these examples are standard JavaScript `import` statements. Theoretically, in Vite-based projects, you could omit the file extension (`.jsx` in this case) in the `import` statement. However, it might be a good idea to include the extension since that's in line with standard JavaScript. When importing from third-party packages (like `useState` from the `react` package), no file extension is added though – you just use the package name. `import` and `export` are standard JavaScript keywords that help with splitting related code across multiple files. Things like variables, constants, classes, or functions can be exported via `export` or `export default` so that they can then be used in other files after importing them there.

Note

If the concept of splitting code into multiple files and using `import` and `export` is brand-new to you, you might want to dive into more basic JavaScript resources on this topic first. For example, MDN has an excellent article that explains the fundamentals, which you can find at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules`.

Of course, the components shown in these examples are highly simplified and also contain features that you haven't learned about yet (e.g., `useState()`). However, the general idea of having standalone building blocks that can be combined should be clear.

When working with React, there are two alternative ways to define components:

- **Class-based components** (or "class components"): Components defined via the `class` keyword
- **Functional components** (or "function components"): Components that are defined via regular JavaScript functions

In all the examples covered in this book, components are built as JavaScript functions. As a React developer, you have to use one of these two approaches as React expects components to be functions or classes.

Note

Until late 2018, you had to use class-based components for certain kinds of tasks—specifically, for components that use state internally. (State will be covered in *Chapter 4*, *Working with Events and State*). However, in late 2018, a new concept was introduced: **React Hooks**. This allows you to perform all operations and tasks with functional components. Consequently, while still supported by React, class-based components are on their way out and are not covered in this book.

In the examples above, there are a couple of other noteworthy things:

- The component functions carry capitalized names (e.g., `SubmitButton`)
- Inside the component functions, other "inner" functions can be defined (e.g., `handleSubmit`, typically written in **camelCase**)
- The component functions return *HTML-like* code (JSX code)
- Features like `useState()` can be used inside the component functions
- The component functions are exported (via `export default`)
- Certain features (like `useState` or the custom component `SubmitButton`) are imported via the `import` keyword

The following sections will take a closer look at these different concepts that make up components and their code.

What Exactly Are Component Functions?

In React, components are functions (or classes, but as mentioned above, those aren't relevant anymore).

A function is a regular JavaScript construct, not a React-specific concept. This is important to note. React is a JavaScript library and consequently **uses JavaScript features** (like functions); React is **not a brand-new programming language**.

When working with React, regular JavaScript functions can be used to encapsulate HTML (or, to be more precise, JSX) code and JavaScript logic that belongs to that markup code. However, it depends on the code you write in a function whether it qualifies to be treated as a React component or not. For example, in the code snippets above, the `handleSubmit` function is also a regular JavaScript function, but it's not a React component. The following example shows another regular JavaScript function that doesn't qualify as a React component:

```
function calculate(a, b) {
  return {sum: a + b};
};
```

Indeed, a function will be treated as a component and can therefore be used like an HTML element in JSX code if it returns a **renderable** value (typically JSX code). This is very important. You can only use a function as a React component in JSX code if it is a function that returns something that can be rendered by React. The returned value technically doesn't have to be JSX code, but in most cases, it will be. You will see an example of non-JSX code being returned in *Chapter 7*, *Portals and Refs*.

In the code snippet where functions named `SubmitButton` and `AuthForm` were defined, those two functions qualified as React components because they both returned JSX code (which is code that can be rendered by React, making it renderable). Once a function qualifies as a React component, it can be used like an HTML element inside of JSX code, just as `<SubmitButton />` was used like a (self-closing) HTML element.

When working with vanilla JavaScript, you, of course, typically call functions to execute them. With functional components, that's different. React calls these functions on your behalf, and for that reason, as a developer, you use them like HTML elements inside of this JSX code.

Note

When referring to renderable values, it is worth noting that by far the most common value type being returned or used is indeed JSX code—i.e., markup defined via JSX. This should make sense because, with JSX, you can define the HTML-like structure of your content and user interface.

But besides JSX markup, there are a couple of other key values that also qualify as renderable and therefore could be returned by custom components (instead of JSX code). Most notably, you can also return strings or numbers as well as arrays that hold JSX elements or strings or numbers.

What Does React Do with All These Components?

If you follow the trail of all components and their `import` and `export` statements to the top, you will find a `root.render(...)` instruction in the main entry script of the React project. Typically, this main entry script can be found in the `main.jsx` file, located in the project's `src/` folder. This `render()` method, which is provided by the React library (to be precise, by the `react-dom` package), takes a snippet of JSX code and interprets and executes it for you.

The complete snippet you find in the root entry file (`main.jsx`) typically looks like this:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

import './index.css';
import App from './App.jsx';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

The exact code you find in your new React project might look slightly different.

It may, for instance, include an extra `<StrictMode>` element that's wrapped around `<App>`. `<StrictMode>` turns on extra checks that can help catch subtle bugs in your React code. But it can also lead to confusing behavior and unexpected error messages, especially when experimenting with React or learning React. As this book is primarily interested in the coverage of React core features and key concepts, `<StrictMode>` will not be used.

While omitted here, strict mode will be covered in *Chapter 10*, *Behind the Scenes of React and Optimization Opportunities*. If you want to learn more about it right now, you can delve into the official documentation: `https://react.dev/reference/react/StrictMode`. Just be aware that some of the effects triggered by strict mode will be easier to understand after you've read more of this book.

To follow along smoothly then, cleaning up a newly created `main.jsx` file to look like the code snippet above is a good idea.

The `createRoot()` method instructs React to create a new **entry point**, which will be used to inject the generated user interface into the actual HTML document that will be served to website visitors. The argument passed to `createRoot()` therefore is a pointer to a DOM element that can be found in `index.html`—the single page that will be served to website visitors.

In many cases, `document.getElementById('root')` is used as an argument. This built-in vanilla JavaScript method yields a reference to a DOM element that is already part of the `index.html` document. Hence, as a developer, you must ensure that such an element with the provided `id` attribute value (`root`, in this example) exists in the HTML file into which the React app script is loaded. In a default React project created via `npm create vite@latest`, this will be the case. You can find a `<div id="root">` element in the `index.html` file in the root project folder.

This `index.html` file is a relatively empty file that only acts as a shell for the React app. React just needs an entry point (defined via `createRoot()`), which will be used to attach the generated user interface to the displayed website. The HTML file and its content, as a result, do not directly define the website content. Instead, the file just serves as a starting point for the React application, allowing React to then take over and control the actual user interface.

Once the root entry point has been defined, a method called `render()` can be called on the `root` object created via `createRoot()`:

```
root.render(<App />);
```

This `render()` method tells React which content (i.e., which React component) should be injected into that root entry point. In most React apps, this is a component called `App`. React will then generate appropriate DOM-manipulating instructions to reflect the markup defined via JSX in the `App` component on the actual web page.

This `App` component is a component function that is imported from some other file. In a default React project, the `App` component function is defined and exported in an `App.jsx` file, which is also located in the `src/` folder.

This component, which is handed to `render()` (`<App />`, typically), is also called the **root component** of the React app. It's the main component that is rendered to the DOM. All other components are nested in the JSX code of that `App` component or the JSX code of even more nested descendent components. You can think of all these components building up a tree of components that is evaluated by React and translated into actual DOM-manipulating instructions.

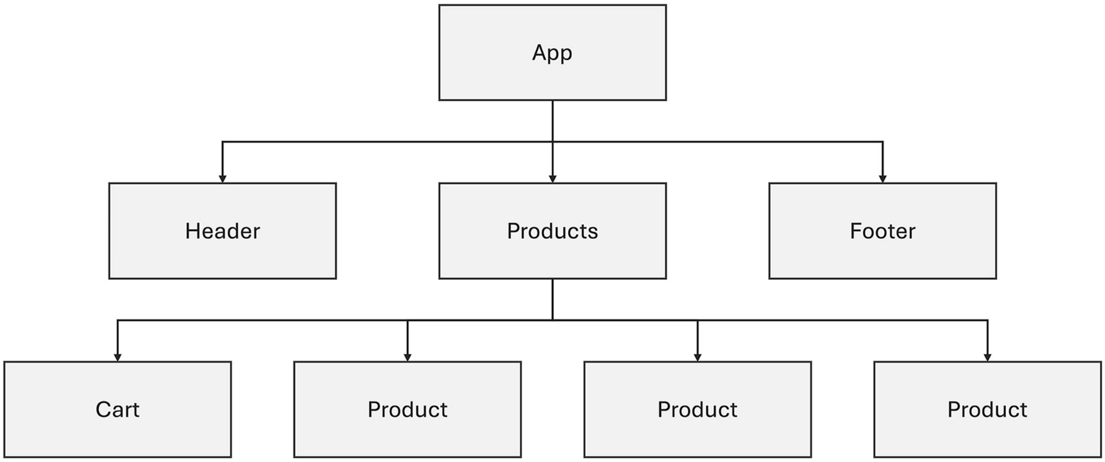


Figure 2.2: Nested React components form a component tree

Note

As mentioned in the previous chapter, React can be used on various platforms. With the `react-native` package, it could be used to build native mobile apps for iOS and Android. The `react-dom` package, which provides the `createRoot()` method (and therefore, implicitly, the `render()` method), is focused on the browser. It provides the "bridge" between React's capabilities and the browser instructions that are required to bring the UI (described via JSX and React components) to life in the browser. If you build for different platforms, replacements for `ReactDOM.createRoot()` and `render()` are required (and, of course, such alternatives do exist).

Either way, no matter whether you use a component function like an HTML element inside of JSX code of other components or use it like an HTML element that's passed as an argument to the `render()` method, React takes care of interpreting and executing the component function on your behalf.

Of course, this is not a new concept. In JavaScript, functions are **first-class objects**, which means that you can pass functions as arguments to other functions. This is basically what happens here, just with the extra twist of using this JSX syntax, which is not a default JavaScript feature.

React executes these component functions for you and translates the returned JSX code into DOM instructions. To be precise, React traverses the returned JSX code and dives into any other custom components that might be used in that JSX code until it ends up with JSX code that is only made up of native, built-in HTML elements (technically, it's not really HTML, but that will be discussed later in this chapter).

Take these two components as an example:

```
function Greeting() {
  return <p>Welcome to this book!</p>;
};

function App() {
  return (
    <div>
      <h2>Hello World!</h2>
      <Greeting />
    </div>
  );
};

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

The `App` component uses the `Greeting` component inside its JSX code. React will traverse the entire JSX markup structure and derive this final JSX code:

```
root.render((
  <div>
    <h2>Hello World!</h2>
    <p>Welcome to this book!</p>
  </div>
), document.getElementById('root'));
```

This code will instruct React and ReactDOM to perform the following DOM operations:

1. Create a `<div>` element
2. Inside that `<div>`, create two child elements: `<h2>` and `<p>`
3. Set the text content of the `<h2>` element to `'Hello World!'`
4. Set the text content of the `<p>` element to `'Welcome to this book!'`
5. Insert the `<div>`, with its children, into the already-existing DOM element, which has the ID `'root'`

This is a bit simplified, but you can think of React handling components and JSX code as described above.

Note

React doesn't actually work with JSX code internally. It's just easier to use as a developer. Later, in this chapter, you will learn what JSX code gets transformed into and what the actual code that React works with looks like.

Built-In Components

As shown in the earlier examples, you can create your own custom components by creating functions that return JSX code. And indeed, that's one of the main things you will do all the time as a React developer: create component functions – lots of component functions.

But, ultimately, if you were to merge all JSX code into just one big snippet of JSX code, as shown in the last example, you would end up with a chunk of JSX code that includes only standard HTML elements like `<div>`, `<h2>`, `<p>`, and so on.

When using React, you don't create brand-new HTML elements that the browser would be able to display and handle. Instead, you create components that **only work inside the React environment**. Before they reach the browser, they have been evaluated by React and "translated" into DOM-manipulating JavaScript instructions (like `document.append(…)`).

But keep in mind that all this JSX code is a feature that's not part of the JavaScript language itself. It's basically **syntactical sugar** (i.e., a simplification regarding the code syntax) provided by the React library and the project setup you're using to write React code. Therefore, elements like `<div>`, when used in JSX code, also **aren't normal HTML elements** because you **don't write HTML code**. It might look like that, but it's inside a `.jsx` file and it's not HTML markup. Instead, it's this special JSX code. It is important to keep this in mind.

Accordingly, these `<div>` and `<h2>` elements you see in all these examples are also just React components in the end. But they are not components built by you, but instead provided by React (or, to be precise, by ReactDOM).

When working with React, you consequently always end up with these primitives—these built-in component functions that are later translated to browser instructions that generate and append or remove normal DOM elements. The idea behind building custom components is to group these elements together such that you end up with reusable building blocks that can be used to build the overall UI. But, in the end, this UI is made up of regular HTML elements.

Note

Depending on your level of frontend web development knowledge, you might have heard about a web feature called **Web Components**. The idea behind this feature is that you can indeed build brand-new HTML elements with vanilla JavaScript.

As mentioned, React does not pick up this feature; you don't build new custom HTML elements with React.

Naming Conventions

All component functions that you can find in this book carry names like `SubmitButton`, `AuthForm`, or `Greeting`.

You can generally name your React functions however you want—at least in the file where you are defining them. But it is a common convention to use the **PascalCase** naming convention, wherein the first character is uppercase and multiple words are grouped into one single word (`SubmitButton` instead of `Submit Button`), where every "subword" then starts with another uppercase character.

In the place where you define your component function, it is only a naming convention, not a hard rule. However, it **is** a hard rule in the place where you **use** the component functions—i.e., in the JSX code where you embed your own custom components.

You can't use your own custom component function as a component like this:

```
<greeting />
```

React forces you to use an uppercase starting character for your own custom component names when using them in JSX code. This rule exists to give React a clear and easy way of telling custom components apart from built-in components like `<div>`, etc. React only needs to look at the starting character to determine whether it's a built-in element or a custom component.

Besides the names of the actual component functions, it is also important to understand file naming conventions. Custom components are typically stored in separate files that live inside a `src/components/` folder. However, this is not a hard rule. The exact placement as well as the folder name is up to you, but it should be somewhere inside the `src/` folder. Using a folder named `components/` is the standard though.

Whereas it is the standard to use PascalCase for the component functions, there is no general default regarding file names. Some developers prefer PascalCase for file names as well; and, indeed, in brand-new React projects, created as described in this book, the `App` component can be found inside a file named `App.jsx`. Nonetheless, you will also encounter many React projects where components are stored in files that follow the kebab-case naming convention. (All lowercase and multiple words are combined into a single word via a dash). With this convention, component functions could be stored in files named `submit-button.jsx`, for example.

Ultimately, it is up to you (and your team) which file naming convention you want to follow. In this book, PascalCase will be used for file names.

JSX vs HTML vs Vanilla JavaScript

As mentioned above, React projects typically contain lots of JSX code. Most custom components will return JSX code snippets. You can see this in all the examples shared thus far, and you will see it in basically every React project you explore, no matter whether you are using React for the browser or other platforms like `react-native`.

But what exactly is this JSX code? How is it different from HTML? And how is it related to vanilla JavaScript?

JSX is a feature that's not part of vanilla JavaScript. What can be confusing, though, is that it's also not directly part of the React library.

Instead, JSX is syntactical sugar that is provided by the build workflow that's part of the overall React project. When you start the development web server via `npm run dev` or build the React app for production (i.e., for deployment) via `npm run build`, you kick off a process that transforms this JSX code back to regular JavaScript instructions. As a developer, you don't see those final instructions but React, the library, actually receives and evaluates them.

So, what does the JSX code get transformed to?

In modern React projects, it gets transformed to rather complex, unintuitive code that looks something like this:

```
function Ld() {
  return St.jsx('p', { children: 'Welcome to this book!' });
}
```

Of course, this code is not very developer-friendly. It's not the kind of code you would write. Instead, it's the code produced by Vite (i.e., by the underlying build process) for the browser to execute.

But you could, in theory, write code like this instead of using JSX—if, for some reason, you wanted to avoid writing JSX code. React has a built-in method you can use instead of JSX: you can use React's `createElement(…)` method.

Here's a concrete example, first in JSX:

```
function Greeting() {
  return <p>Hello World!</p>;
};
```

Instead of using JSX, you could also write this component code like this:

```
function Greeting() {
  return React.createElement('p', {}, 'Hello World!');
};
```

`createElement()` is a method built into the React library. It instructs React to create a paragraph element with `'Hello World!'` as child content (i.e., as inner, nested content). This paragraph element is then created internally first (via a concept called the **virtual DOM**, which will be discussed later in the book, in *Chapter 10*, *Behind the Scenes of React and Optimization Opportunities*). Thereafter, once all elements for all JSX elements have been created, the virtual DOM is translated into real DOM-manipulating instructions that are executed by the browser.

Note

It has been mentioned before that React (in the browser) is actually a combination of two packages: `react` and `react-dom`.

With the introduction of `React.createElement(…)`, it's now easier to explain how these two packages work together: React creates this virtual DOM internally and then passes it to the `react-dom` package. This package then generates the actual DOM-manipulating instructions that must be executed in order to update the web page such that the desired user interface is displayed there.

As mentioned, this will be covered in greater detail in *Chapter 10*.

The middle parameter value (`{}`, in the example) is a JavaScript object that may contain extra configuration for the element that is to be created.

Here's an example where this middle argument becomes important:

```
function Advertisement() {
  return <a href="https://my-website.com">Visit my website</a>;
};
```

This would be transformed to the following:

```
function Advertisement() {
  return React.createElement(
    'a',
    { href: ' https://my-website.com ' },
    'Visit my website'
  );
};
```

The last argument that's passed to `React.createElement(…)` is the child content of the element—i.e., the content that should be between the element's opening and closing tags. For nested JSX elements, nested `React.createElement(…)` calls would be produced:

```
function Alert() {
  return (
    <div>
      <h2>This is an alert!</h2>
    </div>
  );
};
```

This would be transformed like this:

```
function Alert() {
  return React.createElement(
    'div', {}, React.createElement('h2', {}, 'This is an alert!')
  );
};
```

Using React without JSX

Since all JSX code gets transformed to these native JavaScript method calls anyway, you can actually build React apps and user interfaces with React without using JSX.

You can skip JSX entirely if you want to. Instead of writing JSX code in your components and all the places where JSX is expected, you can simply call `React.createElement(…)`.

For example, the following two snippets will produce exactly the same user interface in the browser:

```
function App() {
  return (
    <p>Please visit my <a href="https://my-blog-site.com">Blog</a></p>
  );
};
```

The preceding snippet will ultimately be the same as the following:

```
function App() {
  return React.createElement(
    'p',
    {},
    [
      'Please visit my ',
      React.createElement(
        'a',
        { href: 'https://my-blog-site.com' },
        'Blog'
      )
    ]
  );
};
```

Of course, it's a different question whether you would want to do this. As you can see in this example, it's way more cumbersome to rely on `React.createElement(…)` only. You end up writing a lot more code and deeply nested element structures will lead to code that can become almost impossible to read.

That's why, typically, React developers use JSX. It's a great feature that makes building user interfaces with React way more enjoyable. But it is important to understand that it's neither HTML nor a vanilla JavaScript feature, but that it instead is some syntactical sugar that gets transformed to function calls behind the scenes.

JSX Elements Are Treated Like Regular JavaScript Values

Because JSX is just syntactical sugar that gets transformed, there are a couple of noteworthy concepts and rules you should be aware of:

- JSX elements are just **regular JavaScript values** (functions, to be precise) in the end
- The same rules that apply to all JavaScript values also apply to JSX elements
- As a result, in a place where only one value is expected (e.g., after the `return` keyword), you must only have one JSX element

This code would cause an error:

```
function App() {
  return (
    <p>Hello World!</p>
    <p>Let's learn React!</p>
  );
};
```

The code might look valid at first, but it's actually incorrect. In this example, you would return two values instead of just one. That is not allowed in JavaScript.

For example, the following non-React code would also be invalid:

```
function calculate(a, b) {
  return (
    a + b
    a - b
  );
};
```

You can't return more than one value. No matter how you write it.

Of course, you can return an array or an object though. For example, this code would be valid:

```
function calculate(a, b) {
  return [
    a + b,
    a - b
  ];
};
```

It would be valid because you only return one value: an array. This array contains multiple values, as arrays typically do. That would be fine and the same would be the case if you used JSX code:

```
function App() {
  return [
    <p>Hello World!</p>,
    <p>Let's learn React!</p>
  ];
};
```

This kind of code would be allowed since you are returning one array with two elements inside of it. The two elements are JSX elements in this case, but as mentioned earlier, JSX elements are just regular JavaScript values. Thus, you can use them anywhere where values would be expected.

When working with JSX, you won't see this array approach too often though—simply because it can become annoying to remember wrapping JSX elements via square brackets. It also looks less like HTML, which kind of defeats the purpose and core idea behind JSX (it was invented to allow developers to write HTML code inside JavaScript files).

Instead, if sibling elements are required, as in these examples, a special kind of wrapping component is used: a React **fragment**. That's a built-in component that serves the purpose of allowing you to return or define sibling JSX elements:

```
function App() {
  return (
    <>
      <p>Hello World!</p>
      <p>Let's learn React!</p>
    </>
  );
};
```

This special `<>…</>` element is available in most modern React projects (for instance, ones created via Vite), and you can think of it wrapping your JSX elements with an array behind the scenes. Alternatively, you can also use `<React.Fragment>…</React.Fragment>`. Since some React projects might not support the shorter `<>…</>` syntax, this built-in component is always available.

The parentheses (`()`) that are wrapped around the JSX code in all these examples are required to allow for nice multiline formatting. Technically, you could put all your JSX code into one single line, but that would be pretty unreadable. In order to split the JSX elements across multiple lines, just as you typically do with regular HTML code in `.html` files, you need those parentheses; they tell JavaScript where the returned value starts and ends.

Since JSX elements are regular JavaScript values (after being translated by the build process at least), you can also use JSX elements in all the places where values can be used.

Thus far, that has been the case for all these `return` statements, but you can also store JSX elements in variables or pass them as arguments to other functions:

```
function App() {
  const content = <p>Stored in a variable!</p>; // this works!
  return content;
};
```

This will be important once you dive into slightly more advanced concepts like conditional or repeated content—something that will be covered in *Chapter 5*, *Rendering Lists and Conditional Content*.

JSX Elements Must Have a Closing Tag

Another important rule related to JSX elements is that they must always have a closing tag. Therefore, JSX elements must be self-closing if there is no content between the opening and closing tags:

```
function App() {
  return <img src="some-image.png" />;
};
```

In regular HTML, you would not need that forward backslash at the end. Instead, regular HTML supports void elements (i.e., `<img src="…">`). You can add that forward slash there as well, but it's not mandatory.

When working with JSX, these forward slashes are mandatory if your element doesn't contain any child content.

Moving Beyond Static Content

Thus far, in all these examples, the content that was returned was static. It was content like `<p>Hello World!</p>`—which of course is content that never changes. It will always output a paragraph that says, `'Hello World!'`.

But most websites, of course, need to output dynamic content that may change (e.g., due to user input). Similarly, you'll have a hard time finding lots of websites without any images.

Thus, as a React developer, it's important to know how to output dynamic content (and what "dynamic content" actually means) and how to display images in a React app.

Outputting Dynamic Content

At this point in the book, you don't yet have any tools to make the content more dynamic. To be precise, React requires that state concept (which will be covered in *Chapter 4*, *Working with Events and State*) to change the content that is displayed (e.g., upon user input or some other event).

Nonetheless, since this chapter is about JSX, it is worth diving into the syntax for outputting dynamic content, even though it's not yet truly dynamic:

```
function App() {
  const userName = 'Max';
  return <p>Hi, my name is {userName}!</p>;
};
```

This example technically still produces static output since `userName` never changes, but you can already see the syntax for outputting dynamic content as part of the JSX code. You use opening and closing curly braces (`{…}`) with a JavaScript expression (like the name of a variable or constant, as is the case here) between those braces.

You can put any valid JavaScript expression between those curly braces. For example, you can also call a function (e.g., `{getMyName()}`) or do simple inline calculations (e.g., `{1 + 1}`).

You can't add complex statements like loops or `if` statements between those curly braces though. Again, standard JavaScript rules apply. You output a (potentially) dynamic value, and therefore, anything that produces a single value is allowed in that place. However, it's worth noting that a few value types can't be used for outputting a value in JSX. For example, trying to output a JavaScript object in JSX will cause an error.

It's also worth noting that you're not limited to outputting dynamic content between element tags. Instead, you can also set dynamic values for attributes:

```
function App() {
  const userName = 'Max';
  return <input type="text" value={userName} />;
};
```

Rendering Images

Most websites do not just display plain text. Instead, you often need to render images as well.

Of course, when working with React, you can use the default `<img />` element like in any other web project. But there are two important things to keep in mind when displaying images in React projects:

1. `<img />` must be a self-closing tag.
2. When displaying local images stored inside of the `src/` folder, you must import them into your `.jsx` files.

As explained above, in the *JSX elements must have a closing tag* section, you can't have void JSX elements, i.e., elements without any closing tag.

In addition, when outputting locally stored images (i.e., images stored in the project's `src/` folder, not on some remote server), you typically don't set a string path to the image in your code.

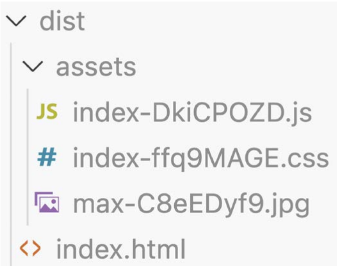
You might be used to outputting images like this:

```
<img src="assets/images/wave.jpg">
```

But React projects (e.g., when created with Vite) do involve some kind of build process. In most projects, the final project structure that will be deployed onto a server will look quite different from the project structure you work on during development.

That being the case, if you store an image in the `src/assets` folder in a Vite-based React project, and you use that as a path (`<img src="src/assets/my-image.jpg" />`), the image will not load on the deployed website. It will not load there because the deployable folder structure will not contain a `src/assets` folder anymore.

Indeed, you can get an idea of the production-ready folder structure by running `npm run build`. This will build the project for deployment and produce a new `dist` folder in your project directory. It's the content of that `dist` folder that will be deployed onto some server. If you inspect that folder, you won't find a `src` folder in there.



dist
  assets
    index-DkiCPOZD.js
    index-ffq9MAGE.css
    max-C8eEDyf9.jpg
  index.html

Figure 2.3: The dist folder contains a different structure

Put in other words: You can't tell the exact path of a locally stored image in advance. That's why you should import the image file into your `.jsx` file. As a result, you'll get a string value that will contain the actual path (which will work in production). This value can then be set as a dynamic value for the `src` attribute of the `<img />` element:

```
import myImage from './assets/my-image.png';

function App() {
  return <img src={myImage} />;
};
```

This might look strange at first, but it is code that will work in pretty much all React projects. Behind the scenes, this import gets analyzed by the underlying build process. The `import` statement then gets removed, and the image path is hardcoded into the production-ready output code (i.e., the code that's stored in the `dist` folder).

There is one important exception though: if you store an image file (or, actually, any asset) in the `public/` folder of your project, you can directly reference its path.

For example, a `demo.jpg` image file stored in `public/images/demo.jpg` can be rendered and displayed like this:

```
function App() {
  return <img src="/images/demo.jpg" />;
};
```

This works because the contents of the `public/` folder are simply copied into the `dist/` folder. Unlike the `src/` folder and its nested files, the `public/` folder files skip the transpilation step.

Please note that the public folder name itself is not part of the paths referenced—it's `src="/images/demo.jpg"`, not `src="/public/images/demo.jpg"`.

Which approach should you use then? Store images in `src/` or `public/`?

For most images, `src/` is a sensible choice since the pre-processing step assigns a unique file name to each imported file. As a result, files can be cached more efficiently once the application is deployed.

Any files imported in the root `index.html` file, or files where the file name must never change (e.g., because it's also referenced by some other app, running on some other server) should typically go into the `public/` folder.

Thus, in most cases, when outputting images that are stored locally in your project, you should store them in the `src/` folder and then import them into your JSX files. When using images that are stored on some remote server, you would instead use the full image URL:

```
function App() {
  return <img src="https://some-server.com/my-image.jpg" />;
};
```

When Should You Split Components?

As you work with React and learn more and more about it, and as you dive into more challenging React projects, you will most likely come up with one very common question: *When should I split a single React component into multiple separate components?*

As mentioned earlier in this chapter, React is all about components, and it is therefore very common to have dozens, hundreds, or even thousands of React components in a single React project.

When it comes to splitting a single React component into multiple smaller components, there is no hard rule you must follow. As mentioned earlier, you could put all your UI code into one single, large component. Alternatively, you could create a separate custom component for every single HTML element and piece of content that you have in your UI. Both approaches are probably not that great. Instead, a good rule of thumb is to create a separate React component for every **data entity** that can be identified.

For example, if you're outputting a "to-do" list, you could identify two main entities: the individual to-do item and the overall list. In this case, it could make sense to create two separate components instead of writing one bigger component.

The advantage of splitting your code into multiple components is that the individual components stay manageable because there's less code per component and component file.

However, when it comes to splitting components into multiple components, a new problem arises: *How do you make your components reusable and configurable?*

```
import Todo from './todo.jsx';

function TodoList() {
  return (
    <ul>
      <Todo />
      <Todo />
    </ul>
  );
};
```

In this example, all "to-dos" would be the same because we use the same `<Todo />` component, which can't be configured. You might want to make it configurable by either adding custom attributes (`<Todo text="Learn React!" />`) or by passing content between the opening and closing tags (`<Todo>Learn React!</Todo>`).

And, of course, React supports this. In the next chapter, you will learn about a key concept called **props**, which allows you to make your components configurable like this.

Summary and Key Takeaways

- React embraces **components**: reusable building blocks that are combined to define the final user interface
- Components must return **renderable** content – typically, JSX code that defines the HTML code that should be produced in the end
- React provides a lot of built-in components: besides special components like `<>…</>`, you get components for all standard HTML elements
- To allow React to tell custom components apart from built-in components, custom component names have to start with capital letters when being used in JSX code (typically, PascalCase naming is used)
- JSX is neither HTML nor a standard JavaScript feature – instead, it's **syntactical sugar** provided by build workflows that are part of all React projects
- You could replace JSX code with `React.createElement(…)` calls, but since this leads to significantly more unreadable code, it's typically avoided
- When using JSX elements, you must not have sibling elements in places where single values are expected (e.g., directly after the `return` keyword)
- JSX elements must always be self-closing if there is no content between the opening and closing tags
- Dynamic content can be output via curly braces (e.g., `<p>{someText}</p>`)
- Images can be rendered by referencing their paths (if stored remotely or in the `public/` folder) or by importing the image files into JSX files and outputting them with the dynamic content syntax
- In most React projects, you split your UI code across dozens or hundreds of components, which are then exported and imported in order to be combined again

What's Next?

In this chapter, you learned a lot about components and JSX. The next chapter builds upon this key knowledge and explains how you can make components reusable by making them configurable.

Before you continue, you can also practice what you have learned up to this point by going through the questions and exercises below.

Test Your Knowledge!

Test your knowledge about the concepts covered in this chapter by answering the questions below. You can then compare your answers to example answers that can be found here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/02-components-jsx/exercises/questions-answers.md`.

1. What's the idea behind using components?
2. How can you create a React component?
3. What turns a regular function into a React component function?
4. Which core rules should you keep in mind regarding JSX elements?
5. How is JSX code handled by React and ReactDOM?

Apply What You Learned

With this and the previous chapter, you have all the knowledge you need to create a React project and populate it with some first, basic components.

Below, you'll find your first two practical activities for this book.

Activity 2.1: Creating a React App to Present Yourself

Suppose you are creating your personal portfolio page, and as part of that page, you want to output some basic information about yourself (e.g., your name or age). You could use React and build a React component that outputs this kind of information, as outlined in the following activity.

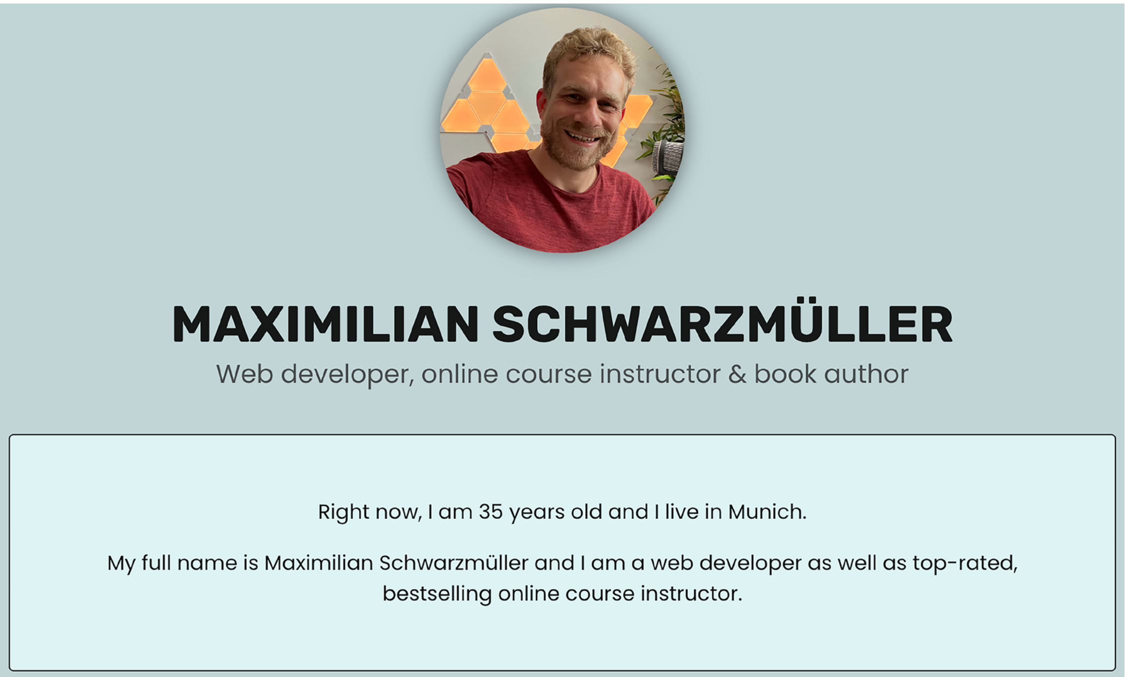
The aim is to create a React app as you learned in the previous chapter (i.e., create it via `npm create vite@latest <your-project-name>` and run `npm run dev` to start the development server) and edit the `App.jsx` file such that you output some basic information about yourself. You could, for example, output your full name, address, job title, or other kinds of information. In the end, it is up to you what content you want to output and which HTML elements you choose.

The idea behind this first exercise is that you practice project creation and working with JSX code.

The steps are as follows:

1. Create a new React project via `npm create vite@latest <project>`. Alternatively, you can use the starting project snapshot provided here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/02-components-jsx/activities/practice-1-start`.
2. Edit the `App.jsx` file in the `/src folder` of the created project and return JSX code with any HTML elements of your choice to output basic information about yourself. You can use the styles in the `index.css` file in the starting project snapshot to apply some styling.
3. Also, store an image in the `src/assets` folder and output it in the `App` component.

You should get output like this in the end:



MAXIMILIAN SCHWARZMÜLLER

Web developer, online course instructor & book author

Right now, I am 35 years old and I live in Munich.

My full name is Maximilian Schwarzmüller and I am a web developer as well as top-rated, bestselling online course instructor.

Figure 2.4: The final activity result—some user information being output on the screen

Note

Styling will of course differ. To get the same styling as shown in the screenshot, use my prepared starting project, which you can find here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/02-components-jsx/activities/practice-1-start`.

Analyze the `index.css` file in that project to determine how to structure your JSX code to apply the styles.

You'll find an example solution on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/02-components-jsx/activities/practice-1/SOLUTION-INSTRUCTIONS.md`.

Besides the linked instructions, you will also find the finished example solution code in the project folder that contains the `SOLUTIONS-INSTRUCTIONS.md` file.

However, before you explore this solution, you should consider trying to solve this task on your own. Even if your result deviates from the example solution, or if you fail to come up with a working application, you'll learn more by at least giving it a try because, as always in life, only practice makes perfect.

Activity 2.2: Creating a React App to Log Your Goals for This Book

Suppose you are adding a new section to your portfolio site, where you plan to track your learning progress. As part of this page, you plan to define and output your main goals for this book (e.g., *"Learn about key React features"*, *"Do all the exercises"*, etc.).

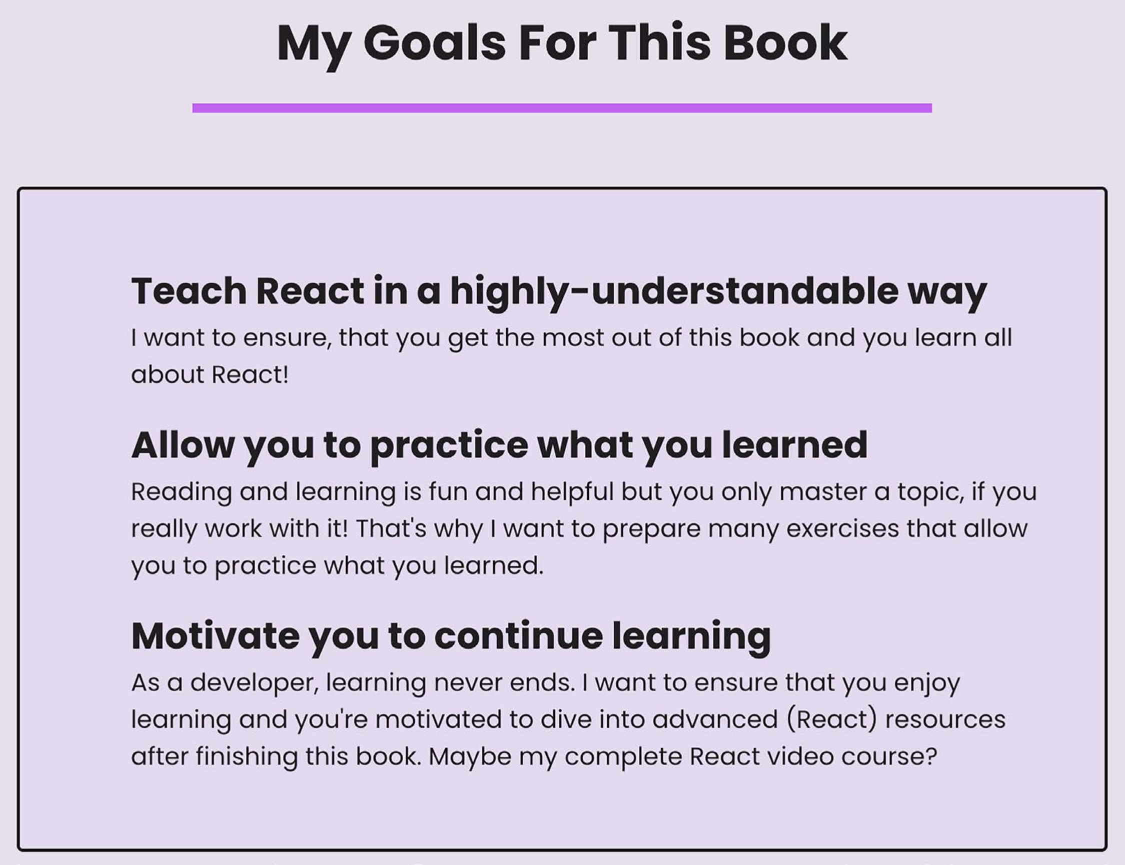
The aim of this activity is to create another new React project in which you add **multiple new components**. Each goal will be represented by a separate component, and all these goal components will be grouped together into another component that lists all the main goals. In addition, you can add an extra header component that contains the main title for the web page.

The steps to complete this activity are as follows:

1. Create a new React project via `npm create vite@latest <project>`, or use the project starting snapshot provided here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/02-components-jsx/activities/practice-2-start`.
2. Inside the new project, create a `components` folder that contains multiple component files (for the individual goals as well as for the list of goals and the page header).
3. Inside the different component files, define and export multiple component functions (`FirstGoal`, `SecondGoal`, `ThirdGoal`, etc.) for the different goals (one component per file).
4. Also, define one component for the overall list of goals (`GoalList`) and another component for the page header (`Header`).
5. In the individual goal components, return JSX code with the goal text and a fitting HTML element structure to hold this content.
6. In the `GoalList` component, import and output the individual goal components.
7. Import and output the `GoalList` and `Header` components in the root `App` component (replace the existing JSX code).

Apply any style of your choice. You can also use the `index.css` file that's part of the starting project snapshot for inspiration.

You should get the following output in the end:



My Goals For This Book

Teach React in a highly-understandable way

I want to ensure, that you get the most out of this book and you learn all about React!

Allow you to practice what you learned

Reading and learning is fun and helpful but you only master a topic, if you really work with it! That's why I want to prepare many exercises that allow you to practice what you learned.

Motivate you to continue learning

As a developer, learning never ends. I want to ensure that you enjoy learning and you're motivated to dive into advanced (React) resources after finishing this book. Maybe my complete React video course?

Figure 2.5: The final page output, showing a list of goals

You'll also find an example solution for this activity on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/02-components-jsx/activities/practice-2/SOLUTION-INSTRUCTIONS.md`.

As before, besides the linked instructions, you will also find the finished example solution code in the project folder that contains the `SOLUTIONS-INSTRUCTIONS.md` file.

3

Components and Props

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Build reusable React components
- Utilize a concept called **props** to make components configurable
- Build flexible user interfaces by combining components with props

Introduction

In the previous chapter, you learned about the key building block of any React-based user interface: **components**. You learned why components matter, how they are used, and how you can build components yourself.

You also learned about JSX, which is the HTML-like markup that's typically returned by component functions. It's this markup that defines what should be rendered on the final web page (in other words, which HTML markup should end up on the final web page that is being served to visitors).

Can Components Do More?

However, so far, those components haven't been too useful. While you could use them to split your web page content into smaller building blocks, the actual reusability of these components was pretty limited. For example, every course goal that you might have as part of an overall course goal list would go into its own component (if you decided to split your web page content into multiple components in the first place).

If you think about it, this isn't too helpful; it would be much better if different list items could share one common component and you just configured that one component with different content or attributes—just like how HTML works.

When writing plain HTML code and describing content with it, you use reusable HTML elements and configure them with different content or attributes. For example, you have one `<a>` HTML element, but thanks to the `href` attribute and the element child content, you can build an endless amount of different anchor elements that point at different resources, as shown in the following snippet:

```
<a href="https://google.com">Use Google</a>
<a href="https://academind.com">Browse Free Tutorials</a>
```

These two elements use the exact same HTML element (`<a>`) but lead to totally different links that would end up on the web page (pointing to two totally different websites).

To fully unlock the potential of React components, it would, therefore, be very useful if you could configure them just like regular HTML elements. And it turns out that you can do exactly that—with another key React concept called **props**.

Using Props in Components

How do you use props in your components? And when do you need them?

The second question will be answered in greater detail a little bit later. For the moment, it's enough to know that you typically will have some components that are reusable and, therefore, need props and some components that are unique and might not need props.

The "how" part of the question is the more important part at this point, and this part can be split into two complementary problems:

1. Passing props to components
2. Consuming props in a component

Passing Props to Components

How would you want props and component configurability to work if you were to design React from the ground up?

Of course, there would be a broad variety of possible solutions, but there is one great role model that can be considered: HTML. As mentioned above, when working with HTML, you pass content and configuration either between element tags or via attributes.

Fortunately, React components work just like HTML elements when it comes to configuring them. Props are simply passed as attributes (to your component) or as child data between component tags, and you can also mix both approaches:

- `<Product id="abc1" price="12.99" />`
- `<FancyLink target="https://some-website.com">Click me</FancyLink>`

For this reason, configuring components is quite straightforward—at least, if you look at them from the consumer's angle (in other words, at how you use them in JSX).

Consuming Props in a Component

How can you get access to the prop values passed into a component, when writing that component's inner code?

Imagine you're building a `GoalItem` component that is responsible for outputting a single goal item (for example, a course goal or project goal) that will be part of an overall goals list.

The parent component JSX markup could look like this:

```
<ul>
  <GoalItem />
  <GoalItem />
  <GoalItem />
</ul>
```

Inside `GoalItem`, the goal (no pun intended) would be to accept different goal titles so that the same component (`GoalItem`) can be used to output these different titles as part of the final list that's displayed to website visitors. Maybe the component should also accept another piece of data (for example, a unique ID that is used internally).

That's how the `GoalItem` component could be used in JSX, as shown in the following example:

```
<ul>
  <GoalItem id="g1" title="Finish the book!" />
  <GoalItem id="g2" title="Learn all about React!" />
</ul>
```

Inside the `GoalItem` component function, the plan would probably be to output dynamic content (in other words, the data received via props) like this:

```
function GoalItem() {
  return <li>{title} (ID: {id})</li>;
}
```

But this component function would not work. It has a problem: `title` and `id` are never defined inside that component function. This code would, therefore, cause an error because you're using a variable that wasn't defined.

Of course, these shouldn't be defined inside the `GoalItem` component anyway, as the idea was to make the `GoalItem` component reusable and receive different `title` and `id` values *from outside the component* (i.e., from the component that renders the list of `<GoalItem>` components).

React provides a solution for this problem: a special parameter value that is passed into every component function automatically by React. This is a special parameter that contains the extra configuration data that is set on the component in JSX code, called the `props` parameter.

The preceding component function could (and should) be rewritten like this:

```
function GoalItem(props) {
  return <li>{props.title} (ID: {props.id})</li>;
}
```

The name of the parameter (`props`) is up to you, but using `props` as a name is a convention because the overall concept is called **props**.

To understand this concept, it is important to keep in mind that these component functions are not called by you somewhere else in your code and that, instead, React will call these functions on your behalf. And since React calls these functions, it can pass extra arguments into them when calling them.

This `props` argument is indeed such an extra argument. React will pass it into every component function, irrespective of whether you defined it as an extra parameter in the component function definition. However, if you didn't define that `props` parameter in a component function, you, of course, won't be able to work with the `props` data in that component.

This automatically provided `props` argument will always contain an object (because React passes an object as a value for this argument), and the properties of this object will be the "attributes" you added to your component (such as the `title` or `id`) inside the JSX code where the component is used.

That's why in this `GoalItem` component example, custom data can be passed via attributes (`<GoalItem id="g1" … />`) and consumed via the `props` object and its properties (`<li>{props.title}</li>`).

Components, Props, and Reusability

Thanks to this props concept, components become *actually* reusable, instead of just being *theoretically* reusable.

Outputting three `<GoalItem>` components without any extra configuration could only render the same goal three times, since the goal text (and any other data you might need) would have to be hardcoded into the component function.

By using props as described above, the same component can be used multiple times with different configurations. That allows you to define some general markup structure and logic once (in the component function) but then use it as often as needed with different configurations.

And if that sounds familiar, that is indeed exactly the same idea that applies to regular JavaScript (or any other programming language) functions. You define logic once, and you can then call it multiple times with different inputs to receive different results. It's the same for components—at least when embracing this props concept.

The Special "children" Prop

It was mentioned before that React passes this `props` object automatically into component functions. That is indeed the case, and as described, this object contains all the attributes you set on the component (in JSX) as properties.

But React does not just package your attributes into this object; it also adds another extra property to the `props` object: the special `children` property (a built-in property whose name is fixed, meaning you can't change it).

The `children` property holds a very important piece of data: the content you might have provided between the component's opening and closing tags.

Thus far, in the examples shown above, the components were mostly self-closing. `<GoalItem id="…" title="…" />` holds no content between the component tags. All the data is passed into the component via attributes.

There is nothing wrong with this approach. You can configure your components with attributes only. But for some pieces of data and some components, it might make more sense and be more logical to actually stick to regular HTML conventions, passing that data between the component tags instead. And the `GoalItem` component is actually a great example.

Which approach looks more intuitive?

1. `<GoalItem id="g1" title="Learn React" />`
2. `<GoalItem id="g1">Learn React</GoalItem>`

You might determine that the second option looks a bit more intuitive and in line with regular HTML because, there, you would also configure a normal list item like this: `<li id="li1">Some list item</li>`.

While you have no choice when working with regular HTML elements (you can't add a `goal` attribute to a `<li>` just because you want to), you do have a choice when working with React and your own components. It simply depends on how you consume props inside the component function. Both approaches can work, depending on the internal component code.

Still, you might want to pass certain pieces of data between component tags, and the special `children` property allows you to do just that. It contains any content you define between the component opening and closing tags. Therefore, in the case of example 2 (in the list above), `children` would contain the string `"Learn React"`.

In your component function, you can work with the `children` value just as you work with any other prop value:

```
function GoalItem(props) {
  return <li>{props.children} (ID: {props.id})</li>;
}
```

Which Components Need Props?

It was mentioned before, but it is extremely important: **props are optional!**

React will always pass **prop** data into your components, but you don't have to work with that `prop` parameter. You don't even have to define it in your component function if you don't plan on working with it.

There is no hard rule that would define which components need **props** and which don't. It comes with experience and simply depends on the role of a component.

You might have a general `Header` component that displays a static header (with a logo, title, and so on), and such a component probably needs no external configuration (in other words, no "attributes" or other kinds of data passed into it). It could be self-contained, with all the required values hardcoded into the component.

But you will also often build and use components like the `GoalItem` component (in other words, components that do need external data to be useful). Whenever a component is used more than once in your React app, there is a high chance that it will utilize props. However, the opposite is not necessarily true. While you will have one-time components that don't use props, you will absolutely also have components that are only used once in the entire app and still take advantage of props. As previously mentioned, it depends on the exact use case and component.

Throughout this book, you will see plenty of examples and exercises that will help you gain a deeper understanding of how to build components and use props.

How to Deal with Multiple Props

As shown in the preceding examples, you are not limited to only one prop per component. Indeed, you can pass and use as many props as your component needs—no matter if that's 1 or 100 (or more) props.

Once you do create components with more than just two or three props, a new question might come up: do you have to add all those props individually (in other words, as separate attributes), or can you pass fewer attributes that contain grouped data, such as arrays or objects?

And indeed, you can. React allows you to pass arrays and objects as prop values as well. In fact, any valid JavaScript value can be passed as a prop value!

This allows you to decide whether you want to have a component with 20 individual props ("attributes") or just one "big" prop. Here's an example of where the same component is configured in two different ways:

```
<Product title="A book" price={29.99} id="p1" />
// or
const productData = {title: 'A book', price: 29.99, id: 'p1'};
<Product data={productData} />
```

Of course, the component must also be adapted internally (in other words, in the component function) to expect either individual or grouped props. But since you're the developer, that is, of course, your choice.

Inside the component function, you can also make your life easier.

There is nothing wrong with accessing prop values via `props.XYZ`, but if you have a component that receives multiple props, repeating `props.XYZ` over and over again could become cumbersome and make the code a bit harder to read.

You can use a default JavaScript feature to improve readability: **object destructuring**.

Object destructuring allows you to extract values from an object and assign those values to variables or constants in a single step:

```
const user = {name: 'Max', age: 29};
const {name, age} = user; // <-- object destructuring in action
console.log(name); // outputs 'Max'
```

You can, therefore, use this syntax to extract all prop values and assign them to equally named variables directly at the start of your component function:

```
function Product({title, price, id}) { // destructuring in action
  … // title, price, id are now available as variables inside this function
}
```

You don't have to use this syntax, but it can make your life easier.

Note

For more information on object destructuring, MDN is a great place to dive deeper. You can access this at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment`.

Spreading Props

Imagine you're building a custom component that should act as a "wrapper" around some other component—a built-in component, perhaps.

For instance, you could be building a custom `Link` component that should return a standard `<a>` element with some custom styling or logic added:

```
function Link({children}) {
  return <a target="_blank" rel="noopener noreferrer">{children}</a>;
};
```

This very simple example component returns a pre-configured `<a>` element. This custom `Link` component configures the anchor element such that new pages are always opened in a new tab. In place of the standard `<a>` element, you could use this `Link` component in your React app to get that behavior out of the box for all your links.

But this custom component suffers from a problem: it's a wrapper around a core element, but by creating your own component, you remove the configurability of that core element. If you were to use this `Link` component in your app, how would you set the `href` prop to configure the link destination?

You might try the following:

```
<Link href="https://some-site.com">Click here</Link>
```

However, this example code wouldn't work because `Link` doesn't accept or use a `href` prop.

Of course, you could adjust the `Link` component function such that a `href` prop is used:

```
function Link({children, href}) {
  return <a href={href} target="_blank" rel="noopener noreferrer">{children}</
a>;
};
```

But what if you also wanted to ensure that the `download` prop could be added if needed?

Well, it's true that you can always accept more and more props (and pass them on to the `<a>` element inside your component), but this reduces the reusability and maintainability of your custom component.

A better solution is to use the standard JavaScript **spread operator** (i.e., the `...` operator) and React's support for that operator when working with components.

For example, the following component code is valid:

```
function Link({children, config}) {
  return <a {...config} target="_blank" rel="noopener noreferrer">{children}</
a>;
};
```

In this example, `config` is expected to be a JavaScript object (i.e., a collection of key-value pairs). The spread operator (`...`), when used in JSX code on a JSX element, converts that object into multiple props.

Consider this example `config` value:

```
const config = { href: 'https://some-site.com', download: true };
```

In this case, when spreading it on `<a>`, (i.e., `<a {…config}>`), the result would be the same as if you had written this code:

```
<a href="https://some-site.com" download={true}>
```

An alternative, more common pattern uses yet another JavaScript feature: the **rest property**. That's a JavaScript pattern that allows you to group properties that have not been destructured into a new object (which then only contains those properties).

```
function Link({children, ...props}) {
  return <a {...props} target="_blank" rel="noopener noreferrer">{children}</
a>;
};
```

In this example, when destructuring props, only the `children` prop is destructured; the other ones are stored in a new object named `props`. The syntax is very similar to the spread operator syntax: you use three dots (`...`). But here, you use the operator in front of the property that should contain all remaining properties. Therefore, it's the place where you use that operator that defines what it does.

You can then use that rest property (`props` in the example) like any other object. In the example above, it's again used to spread its properties as props onto the `<a>` element.

Using this pattern allows you to use the `Link` component in a more natural way, where you don't have to create and use a separate configuration object:

```
<Link href="https://google.com">Can you google that for me?</Link>
```

These behaviors and patterns can be used to build reusable components that should still maintain the configurability of the core element they may be wrapping. This helps you avoid long lists of pre-defined, accepted props and improves the reusability of components.

Prop Chains/Prop Drilling

There is one last phenomenon that is worth noting when learning about props: **prop drilling** or **prop chains**.

It's a problem every React developer will encounter at some point. It occurs when you build a slightly more complex React app that contains multiple layers of nested components that need to send data to each other.

For example, assume that you have a `NavItem` component that should output a navigation link. Inside that component, you might have another nested component, `AnimatedLink`, that outputs the actual link (maybe with some nice animation styling).

The `NavItem` component could look like this:

```
function NavItem(props) {
  return <div><AnimatedLink target={props.target} text="Some text" /></div>;
}
```

And `AnimatedLink` could be defined like this:

```
function AnimatedLink(props) {
  return <a href={props.target}>{props.text}</a>;
}
```

In this example, the `target` prop is passed through the `NavItem` component to the `AnimatedLink` component. The `NavItem` component must accept the `target` prop because it must be passed on to `AnimatedLink`.

That's what prop drilling/prop chains is all about: you forward a prop from a component that doesn't really need it to another component that *does* need it.

Having some prop drilling in your app isn't necessarily bad, and you can definitely accept it. But, if you should end up with longer chains of props (in other words, multiple **pass-through components**), you can use a solution that will be discussed in *Chapter 11*, *Working with Complex States*.

Summary and Key Takeaways

- Props are a key React concept that make components configurable and, therefore, reusable.
- Props are automatically collected and passed into component functions by React.
- You decide (on a per-component basis) whether you want to use the props data (an object) or not.
- Props are passed into components like attributes or, via the special `children` prop, between the opening and closing tags.
- You can use JavaScript features like destructuring, the rest property, or the spread operator to write concise, flexible code.
- Since you are writing the code, it's up to you how you want to pass data via props. Between the tags or as attributes? A single grouped attribute or many single-value attributes? It's up to you.

What's Next?

Props allow you to make components configurable and reusable. Still, they are rather static. Data and, therefore, the UI output doesn't change. You can't react to user events like button clicks.

But the true power of React only becomes visible once you do add events (and reactions to them).

In the next chapter, you will learn how you can add event listeners when working with React, and you will learn how you can react (no pun intended) to events and change the (invisible and visible) state of your application.

Test Your Knowledge!

Test your knowledge regarding the concepts covered in this chapter by answering the following questions. You can then compare your answers to the example answers that can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/03-components-props/exercises/questions-answers.md`:

1. Which "problem" do props solve?
2. How are props passed into components?
3. How are props consumed inside of a component function?
4. Which options exist for passing (multiple) props into components?

Apply What You Learned

With this and the previous chapters, you now have enough basic knowledge to build truly reusable components.

Below, you will find an activity that allows you to apply all the knowledge, including the new props knowledge, you have acquired so far.

Activity 3.1: Creating an App to Output Your Goals for This Book

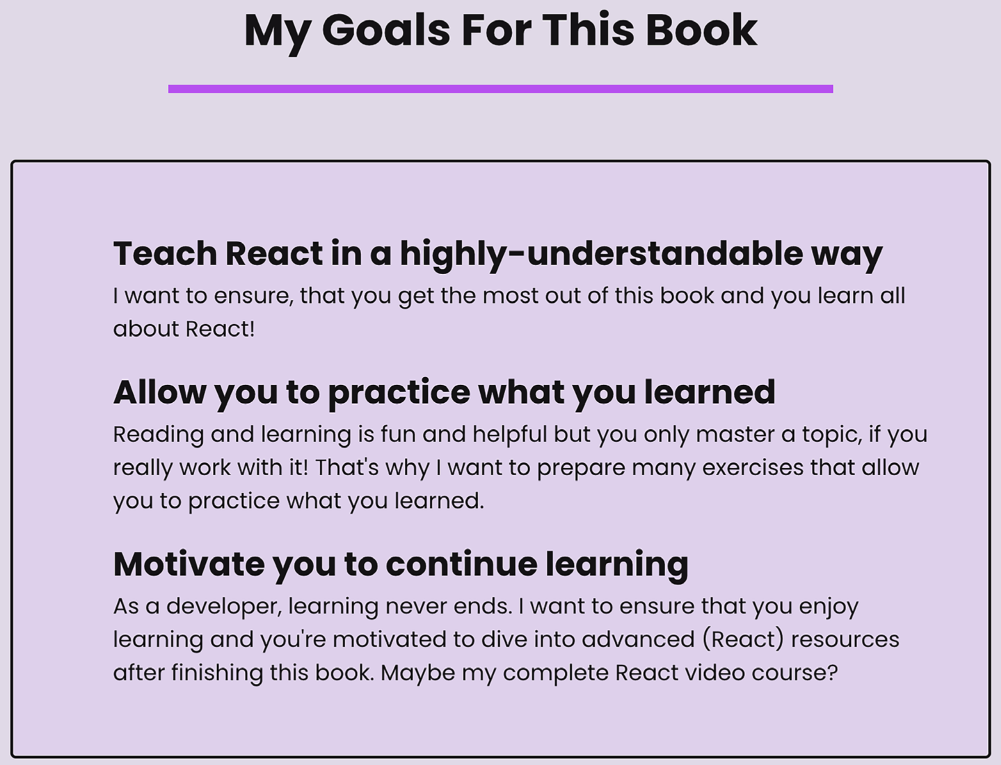
This activity builds upon *Activity 2.2*, *Creating a React App to Log Your Goals for This Book*, from the previous chapter. If you followed along there, you can use your existing code and enhance it by adding props. Alternatively, you can also use the solution provided as a starting point that is accessible at the following link: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/02-components-jsx/activities/practice-2`.

The aim of this activity is to build reusable `GoalItem` components that can be configured via props. Every `GoalItem` component should receive and output a goal title and a short description text, with extra information about the goal.

The steps are as follows:

1. Complete the second activity from the previous chapter.
2. Replace the hardcoded goal item components with a new configurable component.
3. Output multiple goal components with different titles (via props).
4. Set the detailed text description for every goal between the goal component's opening and closing tags.

The final user interface might look like this:



My Goals For This Book

Teach React in a highly-understandable way

I want to ensure, that you get the most out of this book and you learn all about React!

Allow you to practice what you learned

Reading and learning is fun and helpful but you only master a topic, if you really work with it! That's why I want to prepare many exercises that allow you to practice what you learned.

Motivate you to continue learning

As a developer, learning never ends. I want to ensure that you enjoy learning and you're motivated to dive into advanced (React) resources after finishing this book. Maybe my complete React video course?

Figure 3.1: The final result: multiple goals output below each other

Note

You can find a full example solution here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/03-components-props/activities/practice-1`.

4

Working with Events and State

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Add user event handlers (for example, for reacting to button clicks) to React apps
- Update the **user interface** (**UI**) via a concept called **state**
- Build real dynamic and interactive UIs (that is, so that they are not static anymore)

Introduction

In the previous chapters, you learned how to build UIs with the help of React **components**. You also learned about **props**—a concept and feature that enables React developers to build and reuse configurable components.

These are all important React features and building blocks, but with these features alone, you would only be able to build static React apps (that is, web apps that never change). You would not be able to change or update the content on the screen if you only had access to those features. You also would not be able to react to any user events and update the UI in response to such events (for instance, to show an overlay window upon a button click).

Put in other words, you would not be able to build real websites and web applications if you were limited to just components and props.

Therefore, in this chapter, a brand-new concept is introduced: state. State is a React feature that allows developers to update internal data and trigger a UI update based on such data adjustments. In addition, you will learn how to react (no pun intended) to user events such as button clicks or text being entered into input fields.

What's the Problem?

As outlined previously, at this point in the book, there is a problem with all React apps and sites you might be building: they're static. The UI can't change.

To understand this issue a bit better, take a look at a typical React component, as you are able to build it up to this point in the book:

```
function EmailInput() {
  return (
    <div>
      <input placeholder="Your email" type="email" />
      <p>The entered email address is invalid.</p>
    </div>
  );
};
```

This component might look strange though. Why is there a `<p>` element that informs the user about an incorrect email address?

Well, the goal might be to show that paragraph only if the user *did* enter an incorrect email address. That is to say, the web app should wait for the user to start typing and evaluate the user input once the user is done typing (that is, once the input loses focus). Then, the error message should be shown if the email address is considered invalid (for example, an empty input field or a missing @ symbol).

But at the moment, with the React skills picked up thus far, this is something you would not be able to build. Instead, the error message would always be shown since there is no way of changing it based on user events and dynamic conditions. In other words, this React app is a static app, not dynamic. The UI can't change.

Of course, changing UIs and dynamic web apps are things you might want to build. Almost every website that exists contains some dynamic UI elements and features. Therefore, that's the problem that will be solved in this chapter.

How Not to Solve the Problem

How could the component shown previously be made more dynamic?

The following is one solution you could come up with (*spoiler, the code won't work, so you don't need to try running it*):

```
function EmailInput() {
  return (
    <div>
      <input placeholder="Your email" type="email" />
      <p></p>
    </div>
  );
};

const input = document.querySelector('input');
const errorParagraph = document.querySelector('p');

function evaluateEmail(event) {
  const enteredEmail = event.target.value;
  if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
    errorParagraph.textContent = ' The entered email address is invalid.';
  } else {
    errorParagraph.textContent = '';
  }
};

input.addEventListener('blur', evaluateEmail);
```

This code won't work, because you can't select React-rendered DOM elements from inside the same component file this way. This is just meant as a dummy example of how you could try to solve this. That being said, you could put the code below the component function some place where it does execute successfully (for example, into a `setTimeout()` callback that fires after a second, allowing the React app to render all elements onto the screen).

Put in the right place, this code will add the email validation behavior described earlier in this chapter. Upon the built-in `blur` event, the `evaluateEmail` function is triggered. This function receives the `event` object as an argument (automatically, by the browser), and therefore the `evaluateEmail` function is able to parse the entered value from that `event` object via `event.target.value`. The entered value can then be used in an `if` check to conditionally display or remove the error message.

Note

All the preceding code that deals with the `blur` event (such as `addEventListener`) and the `event` object, including the code in the `if` check, is standard JavaScript code. It is not specific to React in any way.

If you find yourself struggling with this non-React code, it's strongly recommended that you dive into more vanilla JavaScript resources (such as the guides on the MDN website at `https://developer.mozilla.org/en-US/docs/Web/JavaScript`) first.

But what's wrong with this code if it would work in some places of the overall application code?

It's imperative code! That means you are writing down step-by-step instructions on what the browser should do. You are not declaring the desired end state; you are instead describing a way of getting there; and it's not using React.

Keep in mind that React is all about controlling the UI and that writing React code is about writing declarative code—instead of imperative code. Revisit *Chapter 2*, *Understanding React Components and JSX*, if that sounds brand new to you.

You could achieve your goal by introducing this kind of code, but you would be working against React and its philosophy (React's philosophy being that you declare your desired end states and let React figure out how to get there). A clear indicator of this is the fact that you would be forced to find the right place for this kind of code in order for it to work.

This is not a philosophical problem, and it's not just some weird hard rule that you should follow. Instead, by working against React like this, you will make your life as a developer unnecessarily hard. You are neither using the tools React gives you nor letting React figure out how to achieve the desired (UI) state.

That does not just mean that you spend time on solving problems you wouldn't have to solve. It also means that you're passing up possible optimizations React might be able to perform under the hood. Your solution is very likely not just leading to more work (that is, more code) for you; it also might result in a buggy result that could also suffer from suboptimal performance.

The example shown previously is a simple one. Think about more complex websites and web apps, such as online shops, vacation rental websites, or web apps such as Google Docs. There, you might have dozens or hundreds of (dynamic) UI features and elements. Managing them all with a mixture of React code and standard vanilla JavaScript code will quickly become a nightmare. Again, refer to *Chapter 2, Understanding React Components and JSX,* of this book to understand the merits of React.

A Better Incorrect Solution

The naïve approach discussed previously doesn't work well. It forces you to figure out how to make the code run correctly (for example, by wrapping parts of it in some `setTimeout()` call to defer execution) and leads to your code being scattered all over the place (that is, inside of React component functions, outside of those functions, and maybe also in totally unrelated files). How about a solution that embraces React, like this:

```
function EmailInput() {
  let errorMessage = '';

  function evaluateEmail(event) {
    const enteredEmail = event.target.value;
    if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
      errorMessage = ' The entered email address is invalid.';
    } else {
      errorMessage = '';
    }
  };

  const input = document.querySelector('input');
```

```
    input.addEventListener('blur', evaluateEmail);

    return (
      <div>
        <input placeholder="Your email" type="email" />
        <p>{errorMessage}</p>
      </div>
    );
  };
```

This code again would not work (even though it's technically valid JavaScript code). Selecting JSX elements doesn't work like this. It doesn't work because `document.querySelector('input')` executes before anything is rendered to the DOM (when the component function is executed for the first time). Again, you would have to delay the execution of that code until the first render cycle is over (you would therefore be once again working against React).

But even though it still would not work, it's closer to the correct solution.

It's closer to the ideal implementation because it embraces React way more than the first attempted solution did. All the code is contained in the component function to which it belongs. The error message is handled via an `errorMessage` variable that is output as part of the JSX code.

The idea behind this possible solution is that the React component that controls a certain UI feature or element is also responsible for its state and events. You might identify two important keywords of this chapter here!

This approach is definitely going in the right direction, but it still wouldn't work for two reasons:

- Selecting the JSX `<input>` element via `document.querySelector('input')` would fail.
- Even if the input could be selected, the UI would not update as expected.

These two problems will be solved next—finally leading to an implementation that embraces React and its features. The upcoming solution will avoid mixing React and non-React code. As you will see, the result will be easier code where you have to do less work (that is, write less code).

Improving the Solution by Properly Reacting to Events

Instead of mixing imperative JavaScript code such as `document.querySelector('input')` with React-specific code, you should fully embrace React and its features.

Since listening to events and triggering actions upon events is an extremely common requirement, React has a built-in solution. You can attach event listeners directly to the JSX elements to which they belong.

The preceding example would be rewritten like this:

```
function EmailInput() {
  let errorMessage = '';

  function evaluateEmail(event) {
    const enteredEmail = event.target.value;
    if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
      errorMessage = 'The entered email address is invalid.';
    } else {
      errorMessage = '';
    }
  };

  return (
    <div>
      <input
        placeholder="Your email"
        type="email"
        onBlur={evaluateEmail} />
      <p>{errorMessage}</p>
    </div>
  );
};
```

This code still will not update the UI, but at least the event is handled properly.

The `onBlur` prop was added to the built-in input element. This prop is made available by React, just as all these base HTML elements (such as `<input>` and `<p>`) are made available as components by React. In fact, all these built-in HTML components come with their standard HTML attributes as React props (plus some extra props, such as the `onBlur` event handling prop).

React exposes all standard events that can be connected to DOM elements as `onXYZ` props (where `XYZ` is the event name, such as `blur` or `click`, starting with a capital character). You can react to the `blur` event by adding the `onBlur` prop. You could listen to a `click` event via the `onClick` prop. You get the idea.

Note

For more information on standard events, see `https://developer.mozilla.org/en-US/docs/Web/Events#event_listing`.

These props require values to fulfill their job. To be precise, they need a pointer to the function that should be executed when the event occurs. In the preceding example, the `onBlur` prop receives a pointer to the `evaluateEmail` function as a value.

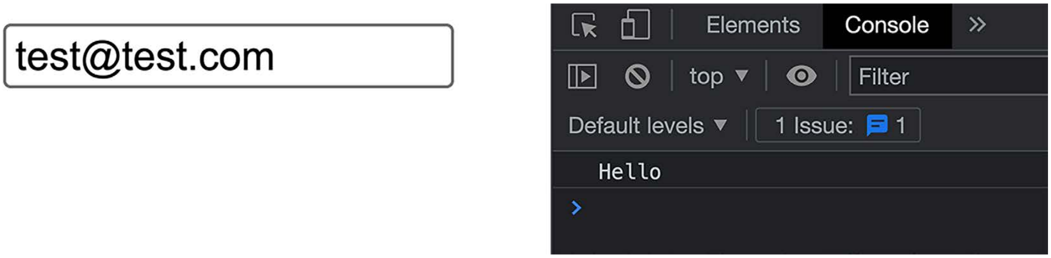
Note

There's a subtle difference between `evaluateEmail` and `evaluateEmail()`. The first is a pointer to the function; the second actually executes the function (and yields the return value, if any). Again, this is not something specific to React but a standard JavaScript concept. If it's not clear, this resource explains it in greater detail: `https://developer.mozilla.org/en-US/docs/Web/Events#event_listing`.

By using these event props, the preceding example code will now finally execute without throwing any errors. You could verify this by adding a `console.log('Hello');` statement inside the `evaluateEmail` function. This will display the `'Hello'` text in the console of your browser developer tools, whenever the input loses focus:

```
function EmailInput() {
  let errorMessage = '';

  function evaluateEmail(event) {
    console.log('Hello');
    const enteredEmail = event.target.value;
    if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
      errorMessage = 'The entered email address is invalid.';
    } else {
      errorMessage = '';
    }
  };

  return (
    <div>
      <input
        placeholder="Your email"
        type="email"
        onBlur={evaluateEmail} />
      <p>{errorMessage}</p>
    </div>
  );
};
```

In the browser console, this looks as follows:



test@test.com

```
Hello
```

Figure 4.1: Displaying some text in the browser console upon removing focus from the input field

This is definitely one step closer to the best possible implementation, but it also still won't produce the desired result of updating the page content dynamically.

Updating State Correctly

By now, you understand how to correctly set up event listeners and execute functions upon certain events. What's missing is a feature that forces React to update the visible UI on the screen and the content that is displayed to the app users.

That's where React's **state** concept comes into play. Like props, state is a key concept of React, but whereas props are about receiving external data inside a component, state is about managing and updating **internal data**. Most importantly, whenever state is updated, React goes ahead and updates the parts of the UI that are affected by the state change.

Here's how state is used in React (of course, the code will then be explained in detail afterward):

```
import { useState } from 'react';

function EmailInput() {
  const [errorMessage, setErrorMessage] = useState('');

  function evaluateEmail(event) {
    const enteredEmail = event.target.value;
    if (enteredEmail.trim() === '' || !enteredEmail.includes('@')) {
      setErrorMessage('The entered email address is invalid.');
    } else {
      setErrorMessage('');
    }
  };

  return (
    <div>
      <input
```

```
          placeholder="Your email"
          type="email"
          onBlur={evaluateEmail} />
        <p>{errorMessage}</p>
      </div>
    );
  };
```

Compared to the example code discussed earlier in this chapter, this code doesn't look much different. But there is a key difference: the usage of the `useState()` Hook.

Hooks are another key concept of React. These are special functions that can only be used inside of React components (or inside of other Hooks, as will be covered in *Chapter 12, Building Custom React Hooks*). Hooks add special features and behaviors to the React components in which they are used. For example, the `useState()` Hook allows a component (and therefore, implicitly React) to set and manage some state that is tied to this component. React provides various built-in Hooks, and they are not all focused on state management. You will learn about other Hooks and their purposes throughout this book.

The `useState()` Hook is an extremely important and commonly used Hook as it enables you to manage data inside a component, which, when updated, tells React to update the UI accordingly.

That is the core idea behind state management and this state concept: state is data, which, when changed, should force React to re-evaluate a component and update the UI if needed.

Using Hooks, such as `useState()`, is pretty straightforward: you import them from `'react'` and you then call them like a function inside your component function. You call them like a function because, as mentioned, React Hooks are functions—just special functions (from React's perspective).

A Closer Look at useState()

How exactly does the `useState()` Hook work and what does it do internally?

By calling `useState()` inside a component function, you register some data with React. It's a bit like defining a variable or constant in vanilla JavaScript. But there is something special: React will track the registered value internally, and whenever you update it, React will re-evaluate the component function in which the state was registered.

React does this by checking whether the data used in the component changed. Most importantly, React validates whether the UI needs to change because of changed data (for example, because a value is output inside the JSX code). If React determines that the UI needs to change, it goes ahead and updates the real DOM in the places where an update is needed (for example, changing some text that's displayed on the screen). If no update is needed, React ends the component re-evaluation without updating the DOM.

React's internal workings will be discussed in great detail *Chapter 10, Behind the Scenes of React and Optimization Opportunities*.

The entire process starts with calling `useState()` inside a component. This creates a state value (which will be stored and managed by React) and ties it to a specific component. An initial state value is registered by simply passing it as a parameter value to `useState()`. In the preceding example, an empty string (`''`) is registered as a first value:

```
const [errorMessage, setErrorMessage] = useState('');
```

As you can see in the example, `useState()` does not just accept a parameter value. It also returns a value: an array with exactly two elements.

The preceding example uses **array destructuring**, which is a standard JavaScript feature that allows developers to retrieve values from an array and immediately assign them to variables or constants. In the example, the two elements that make up the array returned by `useState()` are pulled out of that array and stored in two constants (`errorMessage` and `setErrorMessage`). You don't have to use array destructuring when working with React or `useState()`, though.

You could also write the code like this instead:

```
const stateData = useState('');
const errorMessage = stateData[0];
const setErrorMessage = stateData[1];
```

This works absolutely fine, but when using array destructuring, the code stays a bit more concise. That's why you typically see the syntax using array destructuring when browsing React apps and examples. You also don't have to use constants; variables (via `let`) would be fine as well. As you will see throughout this chapter and the rest of the book, though, the variables won't be reassigned, so using constants makes sense (but it is not required in any way).

Note

If array destructuring or the difference between variables and constants sounds brand new to you, it's strongly recommended that you refresh your JavaScript basics before progressing with this book. As always, MDN provides great resources for that (see `http://packt.link/3B8Ct` for array destructuring, `https://packt.link/hGjqL` for information on the `let` variable, and `https://packt.link/TdPPS` for guidance on the use of `const`).

As mentioned before, `useState()` returns an array with exactly two elements. It will always be exactly two elements—and always exactly the same kind of elements. The first element is always the current state value, and the second element is a function that you can call to set the state to a new value.

But how do these two values (the state value and the state-updating function) work together? What does React do with them internally? How are these two array elements used (by React) to update the UI?

A Look Under the Hood of React

React manages the state values for you, in some internal storage that you, the developer, can't directly access. Since you often do need access to a state value (for instance, some entered email address, as in the preceding example), React provides a way of reading state values: the first element in the array returned by `useState()`. The first element of the returned array holds the current state value. You can therefore use this element in any place where you need to work with the state value (for example, in the JSX code to output it there).

In addition, you often also need to update the state—for example, because a user entered a new email address. Since you don't manage the state value yourself, React gives you a function that you can call to inform React about the new state value. That's the second element in the returned array.

In the example shown before, you call `setErrorMessage('Error!')` to set the `errorMessage` state value to a new string (`'Error!'`).

But why is this managed like this? Why not just use a standard JavaScript variable that you can assign and reassign as needed?

Because React must be informed whenever there's a state that impacts the UI changes. Otherwise, the visible UI doesn't change at all, even in cases where it should. React does not track regular variables and changes to their values, so they have no influence on the state of the UI.

The state-updating function exposed by React (that second array element returned by `useState()`) *does* trigger some internal UI-updating effect though. This state-updating function does more than set a new value; it also informs React that a state value changed and that the UI might therefore be in need of an update.

So, whenever you call `setErrorMessage('Error!')`, React does not just update the value that it stores internally; it also checks the UI and updates it when needed. UI updates can involve anything from simple text changes up to the complete removal and addition of various DOM elements. Anything is possible there!

React determines the new target UI by rerunning (also called re-evaluating) any component functions that are affected by a state change. That includes the component function that executed the `useState()` function that returned the state-updating function that was called. But it also includes any child components, since an update in a parent component could lead to new state data that's also used by some child components (the state value could be passed to child components via props).

If you need a visual of how all this fits together, consider the following diagram:

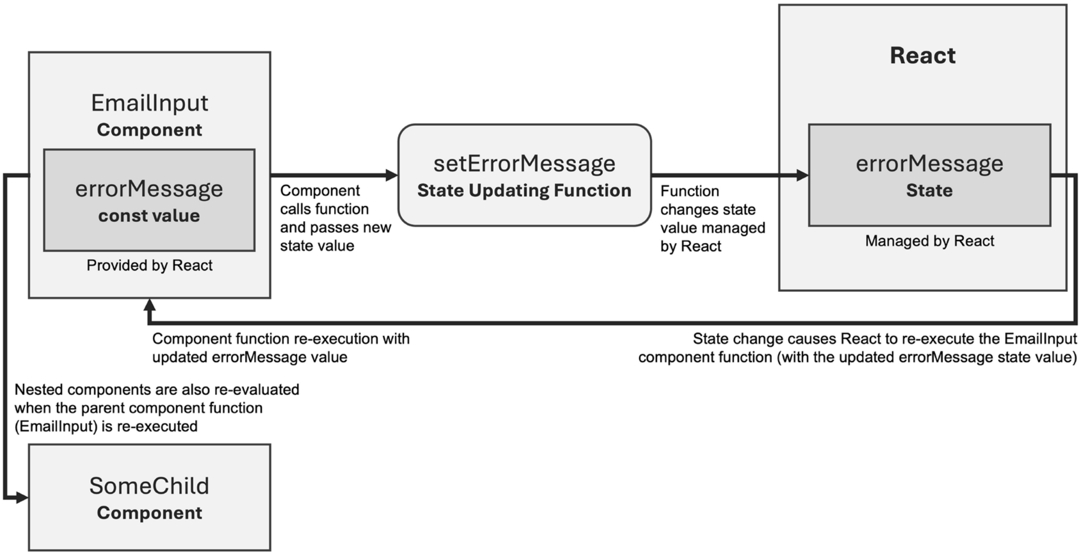

React
EmailInput
Component
errorMessage
const value
Provided by React
Component calls function and passes new state value
setErrorMessage
State Updating Function
Function changes state value managed by React
errorMessage
State
Managed by React
Component function re-execution with updated errorMessage value
State change causes React to re-execute the EmailInput component function (with the updated errorMessage state value)
Nested components are also re-evaluated when the parent component function (EmailInput) is re-executed
SomeChild
Component

Figure 4.2: React state updating flow

It's important to understand and keep in mind that React will re-execute (re-evaluate) a component function if a state-updating function is called in the component function or some parent component function. This also explains why the state value returned by `useState()` (that is, the first array element) can be a constant, even though you can assign new values by calling the state-updating function (the second array element). Since the entire component function is re-executed, `useState()` is also called again (because all the component function code is executed again) and hence a new array with two new elements is returned by React. The first array element is still the current state value.

However, as the component function was called because of a state update, the current state value is now the updated value.

This can be a bit tricky to wrap your head around, but it is how React works internally. In the end, it's just about component functions being called multiple times by React, just as any JavaScript function can be called multiple times.

Naming Conventions

The `useState()` Hook is typically used in combination with array destructuring, like this:

```
const [enteredEmail, setEnteredEmail] = useState('');
```

But when using array destructuring, the names of the variables or constants (`enteredEmail` and `setEnteredEmail`, in this case) are up to you, the developer. Therefore, a valid question is how you should name these variables or constants. Fortunately, there is a clear convention when it comes to React and `useState()`, and these variable or constant names.

The **first element** (that is, the current state value) should be named such that it describes what the state value is all about. Examples would be `enteredEmail`, `userEmail`, `providedEmail`, just `email`, or similar names. You should avoid generic names such as `a` or `value` or misleading names such as `setValue` (which sounds like it is a function—but it isn't).

The **second element** (that is, the state-updating function) should be named such that it becomes clear that it is a function and that it does what it does. Examples would be `setEnteredEmail` or `setEmail`. In general, the convention for this function is to name it `setXYZ`, where `XYZ` is the name you chose for the first element, the current state value variable. (Note, though, that you start with an uppercase character, as in `setEnteredEmail`, not `setenteredEmail`.)

Allowed State Value Types

Managing entered email addresses (or user input in general) is indeed a common use case and example for working with state. However, you're not limited to this scenario and value type.

In the case of entered user input, you will often deal with string values such as email addresses, passwords, blog posts, or similar values. But any valid JavaScript value type can be managed with the help of `useState()`. You could, for example, manage the total sum of multiple shopping cart items—that is, a number—or a Boolean value (for example, *"Did a user confirm the terms of use?"*).

Besides managing primitive value types, you can also store and update reference data types such as objects and arrays.

Note

If the difference between primitive and reference data types is not entirely clear, it's strongly recommended that you dive into this core JavaScript concept before proceeding with this book through the following link: `https://academind.com/tutorials/reference-vs-primitive-values`.

React gives you the flexibility of managing all these value types as state. You can even switch the value type at runtime (just as you can in vanilla JavaScript). It is absolutely fine to store a number as the initial state value and update it to a string at a later point in time.

Just as with vanilla JavaScript, you should, of course, ensure that your program deals with this behavior appropriately, though there's nothing technically wrong with switching types.

Working with Multiple State Values

When building anything but very simple web apps or UIs, you will need multiple state values. Maybe users can not only enter their email but also a username or their address. Maybe you also need to track some error state or save shopping cart items. Maybe users can click a "like" button whose state should be saved and reflected in the UI. There are many values that change frequently and whose changes should be reflected in the UI.

Consider this concrete scenario: you have a component that needs to manage both the value entered by a user into an email input field and the value that was inserted into a password field. Each value should be captured once a field loses focus.

Since you have two input fields that hold different values, you have two state values: the entered email and the entered password. Even though you might use both values together at some point (for example, to log a user in), the values are not provided simultaneously. In addition, you might also need every value to stand alone, since you use it to show potential error messages (for example, *"password too short"*) while the user is entering data.

Scenarios like this are very common, and therefore, you can also manage multiple state values with the `useState()` Hook. There are two main ways of doing that:

1. Use multiple **state slices** (multiple state values)
2. Using one single, *big* state object

Using Multiple State Slices

You can manage multiple state values (also often called **state slices**) by simply calling `useState()` multiple times in your component function.

For the example described previously, a (simplified) component function could look like this:

```
function LoginForm() {
  const [enteredEmail, setEnteredEmail] = useState('');
  const [enteredPassword, setEnteredPassword] = useState('');

  function handleUpdateEmail(event) {
    setEnteredEmail(event.target.value);
  };

  function handleUpdatePassword(event) {
    setEnteredPassword(event.target.value);
  };

  // Below, props are split across multiple lines for better readability
  // This is allowed when using JSX, just as it is allowed in standard HTML
  return (
    <form>
      <input
        type="email"
        placeholder="Your email"
        onBlur={handleUpdateEmail} />
      <input
        type="password"
        placeholder="Your password"
        onBlur={handleUpdatePassword} />
    </form>
  );
};
```

In this example, two state slices are managed by calling `useState()` twice. Therefore, React registers and manages two state values internally. These two values can be read and updated independently from each other.

Note

In the example, the functions that are triggered upon events start with `handle` (`handleUpdateEmail` and `handleUpdatePassword`). This is a convention used by some React developers. Event handler functions start with `handle...` to make it clear that these functions handle certain (user-triggered) events. This is not a convention you have to follow. The functions could have also been named `updateEmail`, `updatePassword`, `emailUpdateHandler`, `passwordUpdateHandler`, or anything else. If the name is meaningful and follows some stringent convention, it's a valid choice.

You can register as many state slices (by calling `useState()` multiple times) as you need in a component. You could have one state value, but you could also have dozens or even hundreds. Typically, though, you will only have a couple of state slices per component since you should try to split bigger components (which might be doing lots of different things) into multiple smaller components to keep them manageable.

The advantage of managing multiple state values like this is that you can update them independently. If the user enters a new email address, you only need to update that email state value. The password state value doesn't matter for your purposes.

A possible disadvantage could be that multiple state slices—and therefore multiple `useState()` calls—lead to lots of lines of code that might bloat your component. As mentioned before, though, you typically should try to break up big components (that handle lots of different slices of state) into multiple smaller components anyway.

Still, there is an alternative to managing multiple state values like this: you can also manage a single, *merged* state value object.

Managing Merged State Objects

Instead of calling `useState()` for every single state slice, you can go for one *big* state object that combines all the different state values:

```
function LoginForm() {
  const [userData, setUserData] = useState({
    email: '',
      password: ''
    });

    function handleUpdateEmail(event) {
      setUserData({
        email: event.target.value,
        password: userData.password
      });
    };

    function handleUpdatePassword(event) {
      setUserData({
        email: userData.email,
        password: event.target.value
      });
    };

    // ... code omitted, because the returned JSX code is the same as before
  };
```

In this example, `useState()` is called only once (i.e., there's only one state slice), and the initial value passed to `useState()` is a JavaScript object. The object contains two properties: `email` and `password`. The property names are up to you, but they should describe the values that will be stored in the properties.

`useState()` still returns an array with exactly two elements. That the initial value is an object does not change anything about that. The first element of the returned array is now just an object instead of a string (as it was in the examples shown earlier). As mentioned before, any valid JavaScript value type can be used when working with `useState()`. Primitive value types such as strings or numbers can be used just as you would reference value types such as objects or arrays (which, technically, are objects of course).

The state-updating function (`setUserData`, in the preceding example) is still a function created by React that you can call to set the state to a new value. Also, you wouldn't have to set it to an object again, though that is typically the default. You don't change value types when updating state unless you have a good reason for doing so (though, technically, you are allowed to switch to a different type at any time).

Note

In the preceding example, the way the state-updating function is used is not entirely correct. It would work but it does violate recommended best practices. You will learn later in this chapter why this is the case and how you should use the state-updating function instead.

When managing state objects as shown in the preceding example, there's one crucial thing you should keep in mind: you must always set all properties the object contains, even the ones that didn't change. This is required because, when calling the state-updating function, you *tell* React which new state value should be stored internally.

Thus, any value you pass as an argument to the state-updating function will overwrite the previously stored value. If you provide an object that contains only the properties that changed, all other properties will be lost since the previous state object is replaced by the new one, which contains fewer properties.

This is a common pitfall and therefore something you must pay attention to. For this reason, in the example shown previously, the property that is not changed is set to the previous state value—for example, `email: userData.email`, where `userData` is the current state snapshot and the first element of the array returned by `useState()`, while setting `password` to `event.target.value`.

It is totally up to you whether you prefer to manage one state value (that is, an object grouping together multiple values) or multiple state slices (that is, multiple `useState()` calls) instead. There is no right or wrong way and both approaches have their advantages and disadvantages.

However, it is worth noting that you should typically try to break up *big* components into smaller ones. Just as regular JavaScript functions shouldn't do too much work in a single function (it is considered a good practice to have separate functions for different tasks), components should focus on one or only a few tasks per component as well. Instead of having a huge `<App />` component that handles multiple forms, user authentication, and a shopping cart directly in one component, it would be preferable to split the code of that component into multiple smaller components that are then combined to build the overall app.

When following that advice, most components shouldn't have too much state to manage anyway, since managing many state values is an indicator of a component doing *too much work*. That's why you might end up using a few state slices per component, instead of large state objects.

Updating State Based on Previous State Correctly

When learning about objects as state values, you learned that it's easy to accidentally overwrite (and lose) data because you might set the new state to an object that contains only the properties that changed—not the ones that didn't. That's why, when working with objects or arrays as state values, it's important to always add the existing properties and elements to the new state value.

Also, in general, setting a state value to a new value that is (at least partially) based on the previous state is a common task. You might set `password` to `event.target.value` but also set `email` to `userData.email` to ensure that the stored email address is not lost due to updating a part of the overall state (that is, because of updating the password to the newly entered value).

That's not the only scenario where the new state value could be based on the previous one, though. Another example would be a `counter` component—for example, a component like this:

```
function Counter() {
  const [counter, setCounter] = useState(0);
  function handleIncrement() {
    setCounter(counter + 1);
  };

  return (
    <>
      <p>Counter Value: {counter}</p>
      <button onClick={handleIncrement}>Increment</button>
    </>
  );
};
```

In this example, a `click` event handler is registered for `<button>` (via the `onClick` prop). Upon every click, the counter state value is incremented by `1`.

This component would work, but the code shown in the example snippet is actually violating an important best practice and recommendation: state updates that depend on some previous state should be performed with the help of a function that's passed to the state-updating function. To be precise, the example should be rewritten like this:

```
function Counter() {
  const [counter, setCounter] = useState(0);

  function handleIncrement() {
    setCounter(function(prevCounter) { return prevCounter + 1; });
    // alternatively, JS arrow functions could be used:
    // setCounter(prevCounter => prevCounter + 1);
  };

  return (
    <>
      <p>Counter Value: {counter}</p>
      <button onClick={handleIncrement}>Increment</button>
    </>
  );
};
```

This might look a bit strange. It might seem like a function is now passed as the new state value to the state-updating function (that is, the number stored in `counter` is replaced with a function). But, indeed, that is not the case.

Technically, a function *is* passed as an argument to the state-updating function, but React won't store that function as the new state value. Instead, when receiving a function as a new state value in the state-updating function, React will call that function for you and pass the latest state value to that function. Therefore, you should provide a function that accepts at least one parameter: the previous state value. This value will be passed into the function automatically by React when React executes the function (which it will do internally).

The function should then also return a value—the new state value that should be stored by React. Also, since the function receives the previous state value, you can now derive the new state value based on the previous state value (for example, by adding the number 1 to it, but any operation could be performed here).

Why is this required if the app worked fine before this change as well? It's required because, in more complex React applications and UIs, React could be processing many state updates simultaneously—potentially triggered from different sources at different times.

When *not* using the approach discussed in the last paragraphs, the order of state updates might not be the expected one and bugs could be introduced into the app. Even if you know that your use case won't be affected and the app does its job without issue, it is recommended to simply adhere to the discussed best practice and pass a function to the state-updating function if the new state depends on the previous state.

With this newly gained knowledge in mind, take another look at an earlier code example:

```
function LoginForm() {
  const [userData, setUserData] = useState({
    email: '',
    password: ''
  });

  function handleUpdateEmail(event) {
    setUserData({
      email: event.target.value,
      password: userData.password
    });
  };

  function handleUpdatePassword(event) {
    setUserData({
      email: userData.email,
      password: event.target.value
    });
  };
  // ... code omitted, because the returned JSX code is the same as before
};
```

Can you spot the error in this code?

It's not a technical error; the code will execute fine, and the app will work as expected. But there is a problem with this code nonetheless. It violates the discussed best practice. In the code snippet, the state in both handler functions is updated by referring to the current state snapshot via `userData.password` and `userData.email`, respectively.

The code snippet should be rewritten like this:

```
function LoginForm() {
  const [userData, setUserData] = useState({
    email: '',
    password: ''
  });

  function handleUpdateEmail(event) {
    setUserData(prevData => ({
      email: event.target.value,
      password: prevData.password
    }));
  };

  function handleUpdatePassword(event) {
    setUserData(prevData => ({
      email: prevData.email,
      password: event.target.value
    }));
  };

  // ... code omitted, because the returned JSX code is the same as before
  // userData is not actively used here, hence you could get a warning
  // regarding that. Simply ignore it or start using userData
  // (e.g., via console.log(userData))
};
```

By passing an arrow function as an argument to `setUserData`, you allow React to call that function. React will do this automatically (that is, if it receives a function in this place, React will call it) and it will provide the previous state (`prevState`) automatically. The returned value (the object that stores the updated `email` or `password` and the currently stored `email` or `password`) is then set as the new state. The result, in this case, might be the same as before, but now the code adheres to recommended best practices.

Note

In the previous example, an arrow function was used instead of a “regular” function. Both approaches are fine, though. You can use either of the two function types; the result will be the same.

In summary, you should always pass a function to the state-updating function if the new state depends on the previous state. Otherwise, if the new state depends on some other value (for instance, user input), directly passing the new state value as a function argument is absolutely fine and recommended.

Two-Way Binding

There is one special usage of React’s state concept that is worth discussing: **two-way binding**.

Two-way binding is a concept that is used if you have an input source (typically an `<input>` element) that sets some state upon user input (for instance, upon the `change` event) and outputs the input at the same time.

Here’s an example:

```
function NewsletterField() {
  const [email, setEmail] = useState('');

  function handleUpdateEmail(event) {
    setEmail(event.target.value);
  };

  return (
    <>
      <input
        type="email"
        placeholder="Your email address"
        value={email}
        onChange={handleUpdateEmail} />
    </>
  );
};
```

Compared to the other code snippets and examples, the difference here is that the component does not just store the user input (upon the `change` event, in this case) but that the entered value is also output in the `<input>` element (via the default `value` prop) thereafter.

This might look like an infinite loop, but React deals with this and ensures that it doesn’t become one. Instead, this is what’s commonly referred to as two-way binding as a value is both set and read from the same source.

You may wonder why this is being discussed here, but it is important to know that it is perfectly valid to write code like this. Also, this kind of code could be necessary if you don't just want to set a value (in this case, the `email` value) upon user input in the `<input>` field but also from other sources. For example, you might have a button in the component that, when clicked, should clear the entered email address.

It might look like this:

```
function NewsletterField() {
  const [email, setEmail] = useState('');

  function handleUpdateEmail(event) {
    setEmail(event.target.value);
  };

  function handleClearInput() {
    setEmail(''); // reset email input (back to an empty string)
  };

  return (
    <>
      <input
        type="email"
        placeholder="Your email address"
        value={email}
        onChange={handleUpdateEmail} />
      <button onClick={handleClearInput}>Reset</button>
    </>
  );
};
```

In this updated example, the `handleClearInput` function is executed when `<button>` is clicked. Inside the function, the `email` state is set back to an empty string. Without two-way binding, the state would be updated, but the change would not be reflected in the `<input>` element. There, the user would still see their last input. The state reflected on the UI (the website) and the state managed internally by React would be different—a bug you absolutely must avoid.

Deriving Values from State

As you can probably tell by now, state is a key concept in React. State allows you to manage data that, when changed, forces React to re-evaluate a component and, ultimately, the UI.

As a developer, you can use state values anywhere in your component (and in your child components, by passing state to them via props). You could, for example, repeat what a user entered like this:

```
function Repeater() {
  const [userInput, setUserInput] = useState('');

  function handleChange(event) {
    setUserInput(event.target.value);
  };

  return (
    <>
      <input type="text" onChange={handleChange} />
      <p>You entered: {userInput}</p>
    </>
  );
};
```

This component might not be too useful, but it will work, and it does use state.

Often, in order to do more useful things, you will need to use a state value as a basis to derive a new (often more complex) value. For example, instead of simply repeating what the user entered, you could count the number of entered characters and show that information to the user:

```
function CharCounter() {
  const [userInput, setUserInput] = useState('');

  function handleChange(event) {
    setUserInput(event.target.value);
  };

  const numChars = userInput.length;

  return (
    <>
      <input type="text" onChange={handleChange} />
      <p>Characters entered: {numChars}</p>
    </>
  );
};
```

Note the addition of the new `numChars` constant (it could also be a variable, via `let`). This constant is derived from the `userInput` state by accessing the `length` property on the string value that's stored in the `userInput` state.

This is important! You're not limited to working with state values only. You can manage some key value as state (that is, the value that will change) and derive other values based on that state value—such as, in this case, the number of characters entered by the user. Indeed, this is something you will do frequently as a React developer.

You might also be wondering why `numChars` is a constant and outside of the `handleChange` function. After all, that is the function that is executed upon user input (that is, upon every keystroke the user makes).

Keep in mind what you learned about how React handles state internally. When you call the state-updating function (`setUserInput`, in this case), React will re-evaluate the component to which the state belongs. This means that the `CharCounter` component function will be called again by React. All the code in that function is therefore executed again.

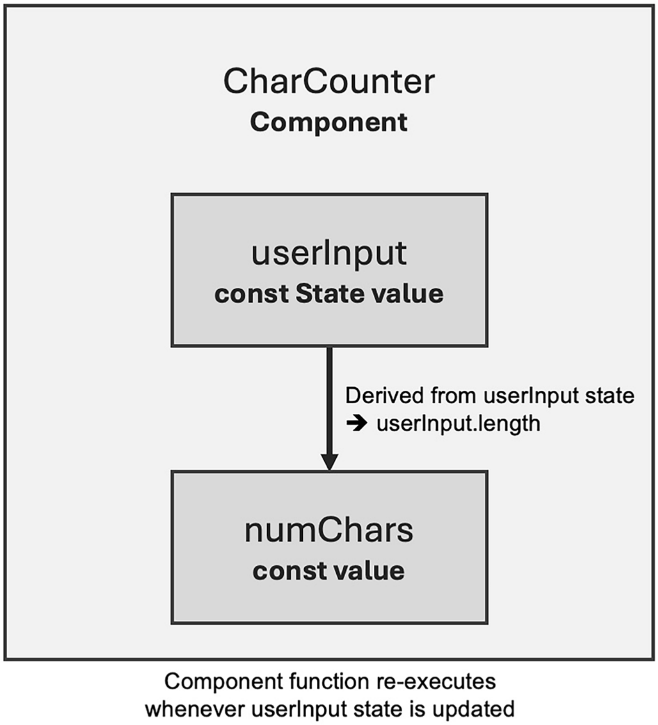

CharCounter
Component
userInput
const State value
Derived from userInput state
➔ userInput.length
numChars
const value
Component function re-executes
whenever userInput state is updated

Figure 4.3: The numChars value is derived from state when the component function executes again

React does re-execute component functions to determine what the UI should look like after the state update; and, if it detects any differences compared to the currently rendered UI, React will go ahead and update the browser UI (that is, the DOM) accordingly. Otherwise, nothing will happen.

Since React calls the component function again, `useState()` will yield its array of values (current state value and state-updating function). The current state value will be the state to which it was set when `setUserInput` was called. Therefore, this new `userInput` value can be used to perform other calculations anywhere in the component function—such as deriving `numChars` by accessing the `length` property of `userInput` (as shown in *Figure 4.3*).

That's why `numChars` can be a constant. For this component execution, it won't be re-assigned. A new value might only be derived when the component function is executed again in the future (that is if `setUserInput` is called again). In that case, a brand-new `numChars` constant would be created (and the old one would be discarded).

Working with Forms and Form Submission

State is commonly used when working with forms and user input. Indeed, most examples in this chapter dealt with some form of user input.

Up to this point, all examples focused on listening to user events that are directly attached to individual input elements. That makes sense because you will often want to listen to events such as keystrokes or an input losing focus. Especially when adding input validation (that is, checking entered values), you might want to use input events to give website users useful feedback while they're typing.

But it's also quite common to react to the overall form submission. For example, the goal could be to combine the input from various input fields and send the data to some backend server. How could you achieve this? How can you listen and react to the submission of a form?

You can do all these things with the help of standard JavaScript events and the appropriate event handler props provided by React. Specifically, the `onSubmit` prop can be added to `<form>` elements to assign a function that should be executed once a form is submitted. To then handle the submission with React and JavaScript, you must ensure that the browser won't do its default thing and generate (and send) an HTTP request automatically.

As in vanilla JavaScript, this can be achieved by calling the `preventDefault()` method on the automatically generated event object.

Here's a full example:

```
function NewsletterSignup() {
  const [email, setEmail] = useState('');
  const [agreed, setAgreed] = useState(false);

  function handleUpdateEmail(event) {
    // could add email validation here
    setEmail(event.target.value);
  };

  function handleUpdateAgreement(event) {
    setAgreed(event.target.checked); // checked is a default JS boolean
property
  };

  function handleSignup(event) {
    event.preventDefault(); // prevent browser default of sending a Http
request

    const userData = {userEmail: email, userAgrees: agreed};
    // doWhateverYouWant(userData);
  };
  return (
    <form onSubmit={handleSignup}>
      <div>
        <label htmlFor="email">Your email</label>
        <input type="email" id="email" onChange={handleUpdateEmail}/>
      </div>
      <div>
        <input type="checkbox" id="agree" onChange={handleUpdateAgreement}/>
        <label htmlFor="agree">Agree to terms and conditions</label>
      </div>
    </form>
  );
};
```

This code snippet handles form submission via the `handleSignup()` function that's assigned to the built-in `onSubmit` prop. User input is still fetched with the help of two state slices (`email` and `agreed`), which are updated upon the inputs' change events.

Note

In the preceding code example, you might've noticed a new prop that wasn't used before in this book: `htmlFor`. This is a special prop, built into React and the core JSX elements it provides. It can be added to `<label>` elements in order to set the `for` attribute for these elements. The reason it is called `htmlFor` instead of just `for` is that, as explained earlier in the book, JSX looks like HTML but isn't HTML. It's JavaScript under the hood. In JavaScript, `for` is a reserved keyword for `for` loops. To prevent problems, the prop is therefore named `htmlFor`.

Using `onSubmit` (combined with `preventDefault()`) for handling form submissions is a very common way of dealing with user input and forms in React. But when working on projects that use React 19 or higher, you can also use an alternative way for handling form submissions: you can use a React feature called **Form Actions**, which will be covered in great detail in *Chapter 9*, *Handling User Input & Forms with Form Actions*.

Lifting State Up

Here's a common scenario and problem: you have two components in your React app and a change or event in component A should change the state in component B. To make this less abstract, consider the following simple example:

```
function SearchBar() {
  const [searchTerm, setSearchTerm] = useState('');

  function handleUpdateSearchTerm(event) {
    setSearchTerm(event.target.value);
  };

  return <input type="search" onChange={handleUpdateSearchTerm} />;
};

function Overview() {
  return <p>Currently searching for {searchTerm}</p>;
};

function App() {
  return (
    <>
      <SearchBar />
      <Overview />
    </>
  );
};
```

In this example, the `Overview` component should output the entered search term. However, the search term is actually managed in another component—namely, the `SearchBar` component. In this simple example, the two components could of course be merged into one single component, and the problem would be solved. But it's very likely that when building more realistic apps, you'll face similar scenarios but with way more complex components. Breaking components up into smaller pieces is considered a good practice since it keeps the individual components manageable.

Having multiple components depend on some shared piece of state is therefore a scenario you will face frequently when working with React.

This problem can be solved by *lifting state up*. When lifting state up, the state is not managed in either of the two components that use it—neither in `Overview`, which reads the state, nor in `SearchBar`, which sets the state—but in a shared ancestor component instead. To be precise, it is managed in the **closest** shared ancestor component. Keep in mind that components are nested into each other and thus a "tree of components" (with the `App` component as the root component) is built up in the end.

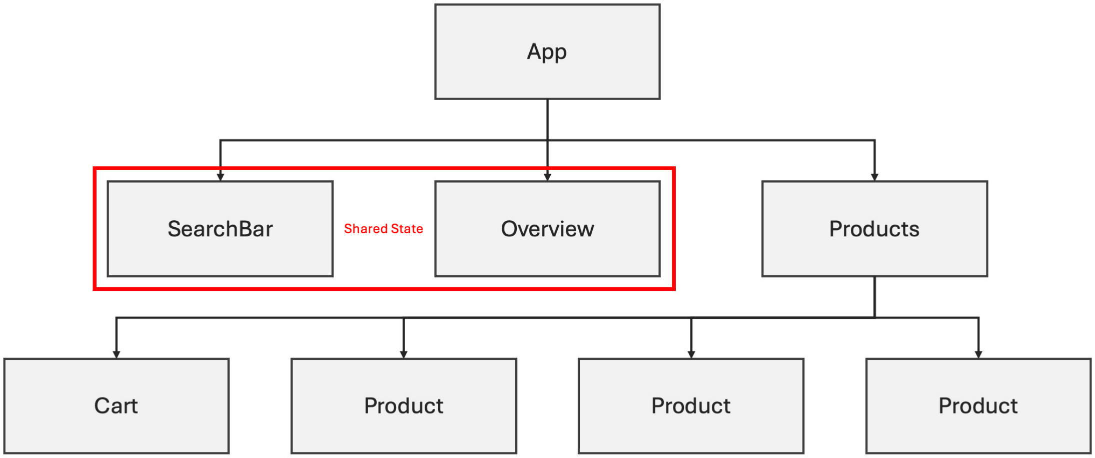


Figure 4.4: An example component tree

In the previous simple code example, the `App` component is the closest (and, in this case, only) ancestor component of both `SearchBar` and `Overview`. If the app was structured as shown in the figure, with state set in one of the `Product` components and used in `Cart`, `Products` would be the closest ancestor component.

State is lifted by using props in the components that need to manipulate (that is, set) or read state, and by registering state in the ancestor component that is shared by the two other components. Here's the updated example from previously:

```
function SearchBar({onUpdateSearch}) {
  return <input type="search" onChange={onUpdateSearch} />;
};

function Overview({currentTerm}) {
  return <p>Currently searching for {currentTerm}</p>;
};

function App() {
  const [searchTerm, setSearchTerm] = useState('');

  function handleUpdateSearchTerm(event) {
    setSearchTerm(event.target.value);
  };

  return (
    <>
      <SearchBar onUpdateSearch={handleUpdateSearchTerm} />
      <Overview currentTerm={searchTerm} />
    </>
  );
};
```

The code didn't actually change that much; it mostly moved around a bit. The state is now managed inside of the shared ancestor and `App` component, and the two other components get access to it via props.

Three key things are happening in this example:

1. The `SearchBar` component receives a prop called `onUpdateSearch`, whose value is a function—a function created in the `App` component and passed down to `SearchBar` from `App`.
2. The `onUpdateSearch` prop is then set as a value to the `onChange` prop on the `<input>` element inside of the `SearchBar` component.
3. The `searchTerm` state (that is, its current value) is passed from `App` to `Overview` via a prop named `currentTerm`.

The first two points could be confusing. But keep in mind that, in JavaScript, functions are first-class objects and regular values. You can store functions in variables and, when using React, pass functions as values for props. Indeed, you could already see that in action at the very beginning of this chapter. When introducing events and event handling, functions were provided as values to all these `onXYZ` props (`onChange`, `onBlur`, and so on).

In this code snippet, a function is passed as a value for a custom prop (that is, a prop expected in a component created by you, not built into React). The `onUpdateSearch` prop expects a function as a value because the prop is then itself being used as a value for the `onChange` prop on the `<input>` element.

The prop is named `onUpdateSearch` to make it clear that it expects a function as a value and that it will be connected to an event. Any name could've been chosen though; it doesn't have to start with `on`. But it's a common convention to name props that expect functions as values and that are intended to be connected to events like this.

Of course, `updateSearch` is not a default event, but since the function will effectively be called upon the `change` event of the `<input>` element, the prop acts like a custom event.

With this structure, the state was lifted up to the `App` component. This component registers and manages the state. However, it also exposes the state-updating function (indirectly, in this case, as it is wrapped by the `handleUpdateSearchTerm` function) to the `SearchBar` component. It also provides the current state value (`searchTerm`) to the `Overview` component via the `currentTerm` prop.

Since the child and descendent components are also re-evaluated by React when state changes in a component, changes in the `App` component will also lead to the `SearchBar` and `Overview` components being re-evaluated. Therefore, the new prop value for `searchTerm` will be picked up, and the UI will be updated by React.

No new React features are needed for this. It's only a combination of state and props. However, depending on how these features are connected and where they are used, both simple and more complex app patterns can be achieved.

Summary and Key Takeaways

- Event handlers can be added to JSX elements via `on[EventName]` props (for example, `onClick`, `onChange`).
- Any function can be executed upon (user) events.
- In order to force React to re-evaluate components and (possibly) update the rendered UI, state must be used.
- State refers to data managed internally by React, and a state value can be defined via the `useState()` Hook.
- React Hooks are JavaScript functions that add special features to React components (for example, the state feature, in this chapter).
- `useState()` always returns an array with exactly two elements:
    - The **first element** is the current state value.
    - The **second element** is a function to set the state to a new value (the *state-updating function*).
- When setting the state to a new value that depends on the previous value, a function should be passed to the state-updating function. This function then receives the previous state as a parameter (which will be provided automatically by React) and returns the new state that should be set.
- Any valid JavaScript value can be set as state—besides primitive values such as strings or numbers. This also includes reference values such as objects and arrays.
- If state needs to change because of some event that occurs in another component, you should *lift the state up* and manage it on a higher, shared level (that is, a common ancestor component).

What's Next?

State is an extremely important building block because it enables you to build truly dynamic applications. With this key concept out of the way, the next chapter will dive into utilizing state (and other concepts learned thus far) to render content conditionally and to render lists of content.

These are common tasks that are required in almost any UI or web app you're building, no matter whether it's about showing a warning overlay or displaying a list of products. The next chapter will help you add such features to your React apps.

Test Your Knowledge!

Test your knowledge about the concepts covered in this chapter by answering the following questions. You can then compare your answers to examples that can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/04-state-events/exercises/questions-answers.md`.

1. Which "problem" does state solve?
2. What's the difference between props and state?
3. How is state registered in a component?
4. Which values does the `useState()` Hook provide?
5. How many state values can be registered for a single component?
6. Does state affect other components (than the component in which it was registered) as well?
7. How should state be updated if the new state depends on the previous state?
8. How can state be shared across multiple components?

Apply What You Learned

With the new knowledge gained in this chapter, you are finally able to build truly dynamic UIs and React applications. Instead of being limited to hardcoded, static content and pages, you can now use state to set and update values and force React to re-evaluate components and the UI.

Here, you will find an activity that allows you to apply all the knowledge, including this new state knowledge, you have acquired up to this point.

Activity 4.1: Building a Simple Calculator

In this activity, you'll build a very basic calculator that allows users to add, subtract, multiply, and divide two numbers with each other.

The steps are as follows:

1. Build the UI by using React components. Be sure to build four separate components for the four math operations, even though lots of code could be reused.
2. Collect the user input and update the result whenever the user enters a value into one of the two related input fields.

   Note that when working with numbers and getting those numbers from user input, you will need to ensure that the entered values are treated as numbers and not as strings.

The final result and UI of the calculator should look like this:

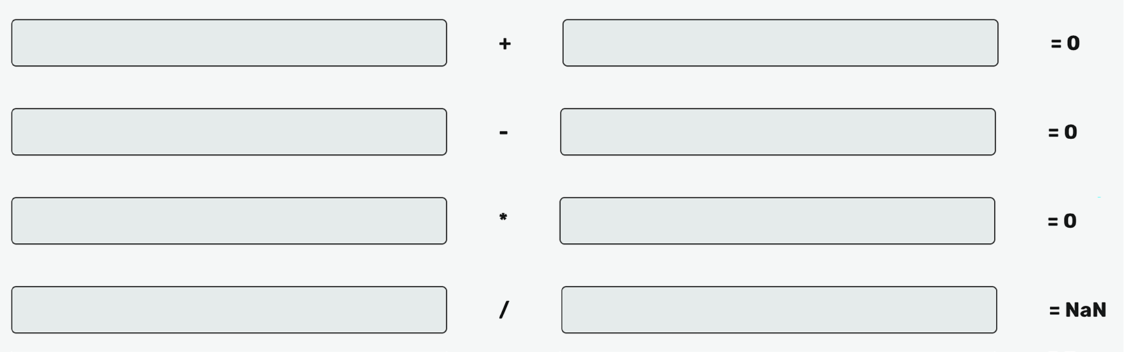


Figure 4.5: Calculator UI

Note

Styling will, of course, differ. To get the same styling as shown in the screenshot, use my prepared starting project, which you can find here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/04-state-events/activities/practice-1-start`.

Analyze the `index.css` file in that project to determine how to structure your JSX code to apply the styles.

Note

You'll find the full example solution here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/04-state-events/activities/practice-1`.

Activity 4.2: Enhancing the Calculator

In this activity, you'll build upon *Activity 4.1* to make the calculator built there slightly more complex. The goal is to reduce the number of components and build one single component in which users can select the mathematical operation via a drop-down element. In addition, the result should be output in a different component—that is, not in the component where the user input is gathered.

The steps are as follows:

1. Remove three of the four components from the previous activity and use one single component for all mathematical operations.
2. Add a drop-down element (`<select>` element) to that remaining component (between the two inputs) and add the four math operations as options (`<option>` elements) to it.
3. Use state to gather both the numbers entered by the user and the math operation chosen via the drop-down (it's up to you whether you prefer one single state object or multiple state slices).

4. Output the result in another component. (Hint: choose a good place for registering and managing the state.)

The result and UI of the calculator should look like this:

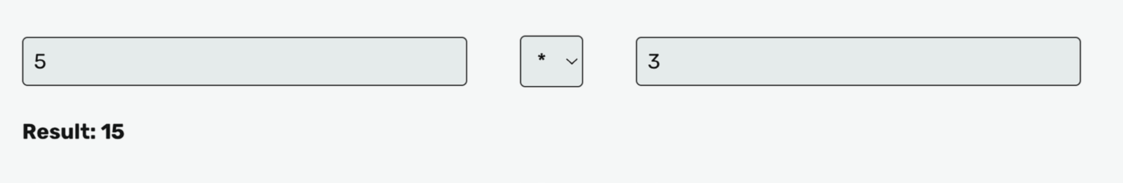


Figure 4.6: UI of the enhanced calculator

Note

You'll find the full example solution here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/04-state-events/activities/practice-2`.

5

Rendering Lists and Conditional Content

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Output dynamic content conditionally
- Render lists of data and map list items to JSX elements
- Optimize lists such that React is able to efficiently update the user interface when needed

Introduction

By this point in the book, you are already familiar with several key concepts, including components, props, state, and events, with which you have all the core tools you need to build all kinds of different React apps and websites. You have also learned how to output dynamic values and results as part of the user interface.

However, there are two topics related to outputting dynamic data that have not yet been discussed in depth: outputting content conditionally and rendering list data. Since most (if not all) websites and web apps you build will require at least one of these two concepts, it is crucial to know how to work with conditional content and list data.

In this chapter, you will therefore learn how to render and display different user interface elements (and even entire user interface sections), based on dynamic conditions. In addition, you will learn how to output lists of data (such as a to-do list with its items) and render JSX elements dynamically for the items that make up a list. This chapter will also explore important best practices related to outputting lists and conditional content.

What Are Conditional Content and List Data?

Before diving into the techniques for outputting conditional content or list data, it is important to understand what exactly is meant by those terms.

Conditional content simply means any kind of content that should only be displayed under certain circumstances. Some examples are as follows:

- Error overlays that should only show up if a user submits incorrect data in a form
- Additional form input fields that appear once the user chooses to enter extra details (such as business details)
- A loading spinner that is displayed while data is sent or fetched to or from a backend server
- A side navigation menu that slides into view when the user clicks on a menu button

This is just a very short list of a few examples. You could, of course, come up with hundreds of additional examples. But it should be clear what all these examples are about in the end: visual elements or entire sections of the user interface that are only shown if certain conditions are met.

In the first example (an error overlay), the condition would be that a user entered incorrect data into a form. The conditionally shown content would then be the error overlay.

Conditional content is extremely common since virtually all websites and web apps have some content that is similar or comparable to the preceding examples.

In addition to conditional content, many websites also output lists of data. It might not always be immediately obvious, but if you think about it, there is virtually no website that does not display some kind of list data. Again, here are some examples of list data that may be outputted on a site:

- An online shop displaying a grid or list of products
- An event booking site displaying a list of events
- A shopping cart displaying a list of cart items
- An orders page displaying a list of orders
- A blog displaying a list of blog posts—and maybe a list of comments below a blog post
- A list of navigation items in the header

An endless list (no pun intended) of examples could be created here. Lists are everywhere on the web. As the preceding examples show, many (probably even most) websites have multiple lists with various kinds of data on the same site.

Take an online shop, for example. Here, you would have a list (or a grid, which is really just another kind of list) of products, a list of shopping cart items, a list of orders, a list of navigation items in the header, and certainly a lot of other lists as well. This is why it is important that you know how to output any kind of list with any kind of data in React-driven user interfaces.

Rendering Content Conditionally

Imagine the following scenario. You have a button that, when clicked, should result in the display of an extra text box, as shown here:

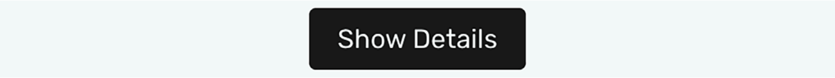


Figure 5.1: Initially, nothing but the button shows up on the screen

After a click on the button, another box is shown:

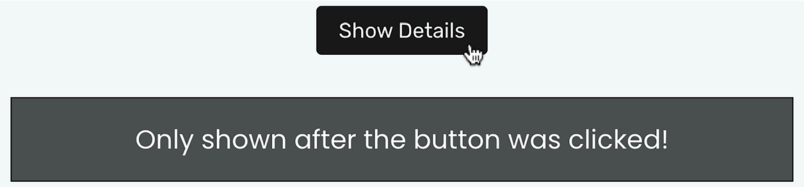


Figure 5.2: After clicking the button, the info box is revealed

This is a very simple example, but not an unrealistic one. Many websites have parts of the user interface that work like this. Showing extra information upon a button click (or some similar interaction) is a common pattern. Just think of nutrition information below a meal on a food order site or an FAQ section where answers are shown after selecting a question.

So, how could this scenario be implemented in a React app?

If you ignore the requirement of rendering some of the content conditionally, the overall React component could look like this:

```
function TermsOfUse() {
  return (
    <section>
      <button>Show Terms of Use Summary</button>
      <p>By continuing, you accept that we will not indemnify you for any
      damage or harm caused by our products.</p>
    </section>
  );
}
```

This component has absolutely no conditional code in it and, therefore, both the button and the extra information box are shown all the time.

In this example, how could the paragraph with the terms-of-use summary text be shown conditionally (that is, only after the button is clicked)?

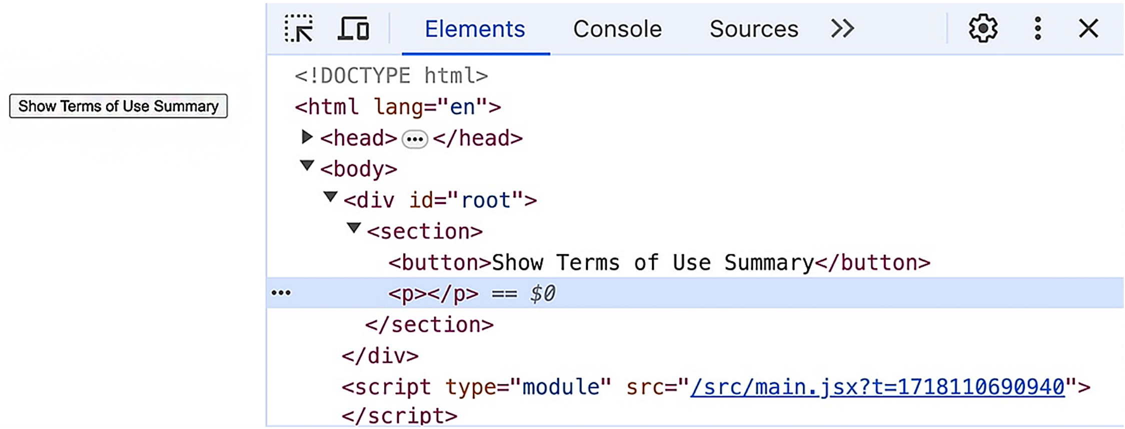
With the knowledge gained throughout the previous chapters, especially *Chapter 4*, *Working with Events and State*, you already have the skills needed to only show the text after the button is clicked. The following code shows how the component could be rewritten to show the full text only after the button is clicked:

```
import { useState } from 'react';
```

```
function TermsOfUse() {
  const [showTerms, setShowTerms] = useState(false);

  function handleShowTermsSummary() {
    setShowTerms(true);
  }

  let paragraphText = '';

  if (showTerms) {
    paragraphText = 'By continuing, you accept that we will not indemnify you for any damage or harm caused by our products.';
  }

  return (
    <section>
      <button onClick={handleShowTermsSummary}>
         Show Terms of Use Summary
      </button>
      <p>{paragraphText}</p>
    </section>
  );
}
```

Parts of the code shown in this snippet already qualify as conditional content. The `paragraphText` value is set conditionally, with the help of an `if` statement based on the value stored in the `showTerms` state.

However, the `<p>` element itself is actually **not** conditional. It is always there, regardless of whether it contains a full sentence or an empty string. If you were to open the browser developer tools and inspect that area of the page, an empty paragraph element would be visible, as shown in the following figure:



```
<!DOCTYPE html>
<html lang="en">
 ▶<head>⋯</head>
 ▼<body>
   ▼<div id="root">
     ▼<section>
         <button>Show Terms of Use Summary</button>
···      <p></p> == $0
       </section>
     </div>
     <script type="module" src="/src/main.jsx?t=1718110690940">
     </script>
```

Figure 5.3: An empty paragraph element is rendered as part of the DOM

Having that empty `<p>` element in the DOM is not ideal. While it's invisible to the user, it's an extra element that needs to be rendered by the browser. The performance impact will very likely be negligible but it's still something you should avoid. A web page doesn't benefit from having empty elements that contain no content.

You can translate your knowledge about conditional values (such as the paragraph text) to conditional elements, however. Besides storing standard values such as text or numbers in variables, you can also store JSX elements in variables. This is possible because, as mentioned in *Chapter 1*, *React – What and Why*, JSX is just syntactic sugar. Behind the scenes, a JSX element is a standard JavaScript function that is executed by React. Also, of course, the return value of a function call can be stored in a variable or constant.

With that in mind, the following code could be used to render the entire paragraph conditionally:

```
import { useState } from 'react';

function TermsOfUse() {
  const [showTerms, setShowTerms] = useState(false);

  function handleShowTermsSummary() {
    setShowTerms(true);
  }

  let paragraph;

  if (showTerms) {
    paragraph = <p>By continuing, you accept that we will not indemnify you for any damage or harm caused by our products.</p>;
  }

  return (
    <section>
      <button onClick={handleShowTermsSummary}>
         Show Terms of Use Summary
      </button>
      {paragraph}
    </section>
  );
}
```

In this example, if `showTerms` is `true`, the `paragraph` variable does not store text but instead an entire JSX element (the `<p>` element). In the returned JSX code, the value stored in the `paragraph` variable is outputted dynamically via `{paragraph}`. If `showTerms` is `false`, `paragraph` stores the value `undefined` and nothing is rendered to the DOM. Therefore, inserting `null` or `undefined` in JSX code leads to nothing being outputted by React. But if `showTerms` is `true`, the complete paragraph is saved as a value and outputted in the DOM.

This is how entire JSX elements can be rendered dynamically. Of course, you are not limited to single elements. You could store entire JSX tree structures (such as multiple, nested, or sibling JSX elements) inside variables or constants. As a simple rule, anything that can be returned by a component function can be stored in a variable.

Different Ways of Rendering Content Conditionally

In the example shown previously, content is rendered conditionally by using a variable, which is set with the help of an `if` statement and then outputted dynamically in JSX code. This is a common (and perfectly fine) technique of rendering content conditionally, but it is not the only approach you can use.

Alternatively, you could also do the following:

- Utilize ternary expressions.
- *Abuse* JavaScript logical operators.
- Use any other valid JavaScript way of selecting values conditionally.

The following sections will explore each approach in detail.

Utilizing Ternary Expressions

In JavaScript (and many other programming languages), you can use **ternary expressions** (also referred to as **conditional ternary operators**) as alternatives to `if` statements. Ternary expressions can save you lines of code, especially with simple conditions where the main goal is to assign some variable value conditionally.

Here is a direct comparison—first starting with a regular `if` statement:

```
let a = 1;
if (someCondition) {
  a = 2;
}
```

Here is the same logic, implemented with a ternary expression:

```
const a = someCondition ? 2 : 1;
```

This is standard JavaScript code, not specific to React. However, it is important to understand this core JavaScript feature in order to understand how it can be used in React apps.

Translated to the previous React example, the paragraph content could be set and outputted conditionally with the help of ternary expressions like this:

```
import { useState } from 'react';

function TermsOfUse() {
  const [showTerms, setShowTerms] = useState(false);

  function handleShowTermsSummary() {
    setShowTerms(true);
  }

  const paragraph = showTerms ? <p>By continuing, you accept that we will not
indemnify you for any damage or harm caused by our products.</p> : null;

  return (
    <section>
      <button onClick={handleShowTermsSummary}>
         Show Terms of Use Summary
      </button>
      {paragraph}
    </section>
  );
}
```

As you can see, the overall code is a bit shorter than before, when an `if` statement was used. The paragraph constant contains either the paragraph (including the text content) or `null`. `null` is used as an alternative value because `null` can safely be inserted into JSX code as it simply leads to nothing being rendered in its place.

A disadvantage of ternary expressions is that readability and understandability may suffer—especially when using nested ternary expressions, like in the following example:

```
const paragraph = !showTerms ? null : someOtherCondition ? <p>By continuing,
you accept that we will not indemnify you for any damage or harm caused by our
products.</p> : null;
```

This code is difficult to read and even more difficult to understand. For this reason, you should typically avoid writing nested ternary expressions and fall back to `if` statements in such situations.

However, despite these potential disadvantages, ternary expressions can help you write less code in React apps, especially when using them inline, directly inside some JSX code:

```
import { useState } from 'react';

function TermsOfUse() {
  const [showTerms, setShowTerms] = useState(false);

  function handleShowTermsSummary() {
    setShowTerms(true);
  }

  return (
    <section>
```

```
        <button onClick={handleShowTermsSummary}>
          Show Terms of Use Summary
        </button>
        {showTerms ? <p>By continuing, you accept that we will not indemnify you 
for any damage or harm caused by our products.</p> : null}
      </section>
    );
  }
```

This is the same example as before, only now it's even shorter since here you avoid using the `paragraph` constant by utilizing the ternary expression directly inside of the JSX snippet. This allows for relatively lean component code, so it is quite common to use ternary expressions in JSX code in React apps to take advantage of this.

Abusing JavaScript Logical Operators

Ternary expressions are popular because they enable you to write less code, which, when used in the right places (and avoiding nesting multiple ternary expressions), can help with overall readability.

Especially in React apps, in JSX code you will often write ternary expressions like this:

```
<div>
  {showDetails ? <h1>Product Details</h1> : null}
</div>
```

Or, like this:

```
<div>
  {showTerms ? <p>Our terms of use …</p> : null}
</div>
```

What do these two snippets have in common?

They are unnecessarily long because, in both examples, the else case (`: null`) must be specified, even though it adds nothing to the final user interface. After all, the primary purpose of these ternary expressions is to render JSX elements (`<h1>` and `<p>`, in the preceding examples). The `else` case (`: null`) simply means nothing is rendered if the conditions (`showDetails` and `showTerms`) are not met.

This is why a different pattern is popular among React developers:

```
<div>
  {showDetails && <h1>Product Details</h1>}
</div>
```

This is the shortest possible way of achieving the intended result, rendering only the `<h1>` element and its content if `showDetails` is `true`.

This code uses (or abuses) an interesting behavior of JavaScript's logical operators, specifically of the `&&` (logical and) operator. In JavaScript, the `&&` operator returns the second value (that is, the value after `&&`) if the first value (that is, the value before `&&`) is `true` or truthy (that is, not `false`, `undefined`, `null`, `0`, and so on). Normally, you'd use the `&&` operator in `if` statements or ternary expressions. However, when working with React and JSX, you can take advantage of the behavior described previously to output truthy values conditionally. This technique is also called **short-circuiting**.

For example, the following code would output `'Hello'`:

```
console.log(1 === 1 && 'Hello');
```

This behavior can be used to write very short expressions that check a condition and then output another value, as shown in the preceding example.

> **Note**
>
> It is worth noting that using `&&` can lead to unexpected results if you're using it with non-Boolean condition values (that is, if the value in front of `&&` holds a non-Boolean value). If `showDetails` were `0` instead of `false` (for whatever reason), the number `0` would be displayed on the screen. You should therefore ensure that the value acting as a condition yields `null` or `false` instead of arbitrary falsy values. You could, for example, force a conversion to a Boolean by adding `!!` (for example, `!!showDetails`). That is not required if your condition value already holds `null` or `false`.

Get Creative!

At this point, you have learned about three different ways of defining and outputting content conditionally (regular `if` statements, ternary expressions, and using the `&&` operator). However, the most important point is that React code is ultimately just regular JavaScript code. Hence, any approach that selects values conditionally will work.

If it makes sense in your specific use case and React app, you could also have a component that selects and outputs content conditionally like this:

```
const languages = {
  de: 'de-DE',
  us: 'en-US',
  uk: 'en-GB'
};

function LanguageSelector({country}) {
  return <p>Selected Language: {languages[country]}</p>
}
```

This component outputs either `'de-DE'`, `'en-US'`, or `'en-GB'` based on the value of the `country` prop. This result is achieved by using JavaScript's dynamic property selection syntax. Instead of selecting a specific property via the dot notation (such as `person.name`), you can select property values via the bracket notation. With that notation, you can either pass a specific property name (`languages['de-DE']`) or an expression that yields a property name (`languages[country]`).

Selecting property values dynamically like this is another common pattern for picking values from a map of values. It is therefore an alternative to specifying multiple `if` statements or ternary expressions.

Also, in general, you can use any approach that works in standard JavaScript—because React is, after all, just standard JavaScript at its core.

Which Approach is Best?

Various ways of setting and outputting content conditionally have been discussed, but which approach is best?

That really is up to you (and, if applicable, your team). The most important advantages and disadvantages have been highlighted, but ultimately, it is your decision. If you prefer ternary expressions, there's nothing wrong with choosing them over the logical `&&` operator, for example.

It will also depend on the exact problem you are trying to solve. If you have a map of values (such as a list of countries and their country language codes), going for dynamic property selection instead of multiple `if` statements might be preferable. On the other hand, if you have a single `true`/`false` condition (such as `age > 18`), using a standard `if` statement or the logical `&&` operator might be best.

Setting Element Tags Conditionally

Outputting content conditionally is a very common scenario. But sometimes, you will also want to choose the type of HTML tag that will be outputted conditionally. Typically, this will be the case when you build components whose main task is to wrap and enhance built-in components.

Here's an example:

```
function Button({isButton, config, children}) {
  if (isButton) {
    return <button {...config}>{children}</button>;
  }
  return <a {...config}>{children}</a>;
};
```

This `Button` component checks whether the `isButton` prop value is truthy and, if that is the case, returns a `<button>` element. The `config` prop is expected to be a JavaScript object, and the standard JavaScript spread operator (`...`) is used to then add all key-value pairs of the `config` object as props to the `<button>` element. If `isButton` is not truthy (maybe because no value was provided for `isButton`, or because the value is `false`), the `else` condition becomes active. Instead of a `<button>` element, an `<a>` element is returned.

Note

Using the spread operator (`...`) to translate an object's properties (key-value pairs) into component props is another common React pattern (and was introduced in *Chapter 3*, *Components and Props*). The spread operator is not a React-specific operator but using it for this special purpose *is*.

When spreading an object such as `{link: 'https://some-url.com', isButton: false}` onto an `<a>` element (via `<a {...obj}>`), the result would be the same as if all props had been set individually (that is, `<a link="https://some-url.com" isButton={false}>`).

This pattern is particularly popular in situations where you build custom *wrapper components* that wrap a common core component (e.g., `<button>`, `<input>`, or `<a>`) to add certain styles or behaviors, while still allowing the component to be used in the same way as the built-in component (that is, you can set all the default props).

The `Button` component from the preceding example returns two totally different JSX elements, depending on the `isButton` prop value. This is a great way of checking a condition and returning different content (that is, conditional content).

However, by using a special React behavior, this component could be written with even less code:

```
function Button({isButton, config, children}) {
  const Tag = isButton ? 'button' : 'a';
  return <Tag {...config}>{children}</Tag>;
};
```

The special behavior is that tag names can be stored (as string values) in variables or constants, and that those variables or constants can then be used like JSX elements in JSX code (as long as the variable or constant name starts with an uppercase character, like all your custom components).

The `Tag` constant in the preceding example stores either the `'button'` or `'a'` string. Since it starts with an uppercase character (`Tag`, instead of `tag`), it can then be used like a custom component inside of JSX code snippets. React accepts this as a component, even though it isn't a component function. This is because a standard HTML element tag name is stored, so React can render the appropriate built-in component. The same pattern could also be used with custom components. Instead of storing string values, you would store pointers to your custom component functions through the following:

```
import MyComponent from './my-component.jsx';
import MyOtherComponent from './my-other-component.jsx';

const Tag = someCondition ? MyComponent : MyOtherComponent;
```

This is another useful pattern that can help save code and hence leads to leaner components.

Outputting List Data

Besides outputting conditional data, you will often work with list data that should be outputted on a page. As mentioned earlier in this chapter, some examples are lists of products, transactions, and navigation items.

Typically, in React apps, such list data is received as an array of values. For example, a component might receive an array of products via props (passed into the component from inside another component that might be getting that data from some backend API):

```
function ProductsList({products}) {
  // … todo!
};
```

In this example, the products array could look like this:

```
const products = [
  {id: 'p1', title: 'A Book', price: 59.99},
  {id: 'p2', title: 'A Carpet', price: 129.49},
  {id: 'p3', title: 'Another Book', price: 39.99},
];
```

This data can't be outputted like this, though. Instead, the goal is typically to translate it into a list of JSX elements that fits. For example, the desired result could be the following:

```
<ul>
  <li>
    <h2>A Book</h2>
    <p>$59.99</p>
  </li>
  <li>
    <h2>A Carpet</h2>
    <p>$129.49</p>
  </li>
  <li>
    <h2>Another Book</h2>
    <p>$39.99</p>
  </li>
</ul>
```

How can this transformation be achieved?

Again, it's a good idea to ignore React and find a way to transform list data with standard JavaScript. One possible way to achieve this would be to use a `for…of` loop, as shown:

```
const transformedProducts = [];
for (const product of products) {
    transformedProducts.push(product.title);
  }
```

In this example, the list of product objects (`products`) is transformed into a list of product titles (that is, a list of string values). This is achieved by looping through all product items in `products` and extracting only the `title` property from each product. This `title` property value is then pushed into the new `transformedProducts` array.

A similar approach can be used to transform the list of objects into a list of JSX elements:

```
const productElements = [];
for (const product of products) {
  productElements.push((
    <li>
      <h2>{product.title}</h2>
      <p>${product.price}</p>
    </li>
  ));
}
```

The first time you see code like this, it might look a bit strange. But keep in mind that JSX code can be used anywhere where regular JavaScript values (that is, numbers, strings, objects, and so on) can be used. Therefore, you can also `push` a JSX value onto an array of values. Since it's JSX code, you can also output content dynamically in those JSX elements (such as `<h2>{product.title}</h2>`).

This code is valid and is an important first step toward outputting list data. But it is only the first step, since the current data was transformed but still isn't returned by a component.

How can such an array of JSX elements be returned then?

The answer is that it can be returned without any special tricks or code. JSX actually accepts array values as dynamically outputted values.

You can output the `productElements` array like this:

```
return (
  <ul>
    {productElements}
  </ul>
);
```

When inserting an array of JSX elements into JSX code, all JSX elements inside that array are outputted next to each other. So, the following two snippets would produce the same output:

```
return (
  <div>
    {[<p>Hi there</p>, <p>Another item</p>]}
  </div>
);

return (
  <div>
    <p>Hi there</p>
    <p>Another item</p>
  </div>
);
```

With this in mind, the `ProductsList` component could be written like this:

```
function ProductsList({products}) {
  const productElements = [];
  for (const product of products) {
    productElements.push((
      <li>
        <h2>{product.title}</h2>
        <p>${product.price}</p>
      </li>
    ));
  }

  return (
    <ul>
      {productElements}
    </ul>
  );
};
```

This is one possible approach for outputting list data. As explained earlier in this chapter, it's all about using standard JavaScript features and combining those features with JSX.

However, it's not necessarily the most common way of outputting list data in React apps. In most projects, you'll encounter a different solution.

Mapping List Data

Outputting list data with `for` loops works, as you can see in the preceding examples. However, just as with `if` statements and ternary expressions, you can replace `for` loops with an alternative syntax to write less code and improve component readability.

JavaScript offers a built-in array method that can be used to transform array items: the `map()` method. `map()` is a default method that can be called on any JavaScript array. It accepts a function as a parameter and executes that function for every array item. The return value of this function should be the transformed value. `map()` then combines all these returned, transformed values into a new array that is then returned by `map()`.

You could use `map()` like this:

```
const users = [
  {id: 'u1', name: 'Max', age: 35},
  {id: 'u2', name: 'Anna', age: 32}
];
const userNames = users.map(user => user.name);
// userNames = ['Max', 'Anna']
```

In this example, `map()` is used to transform the array of user objects into an array of usernames (that is, an array of string values).

The `map()` method is often able to produce the same result as that of a `for` loop but with less code.

Therefore, `map()` can also be used to generate an array of JSX elements and the `ProductsList` component from before could be rewritten like this:

```
function ProductsList({products}) {
  const productElements = products.map(product => (
      <li>
        <h2>{product.title}</h2>
        <p>${product.price}</p>
      </li>
    )
  );

  return (
    <ul>
      {productElements}
    </ul>
  );
};
```

This is already shorter than the earlier `for` loop example. However, just as with ternary expressions, the code can be shortened even more by moving the logic directly into the JSX code:

```
function ProductsList({products}) {
  return (
    <ul>
      {products.map(product => (
          <li>
            <h2>{product.title}</h2>
            <p>${product.price}</p>
          </li>
        )
      )}
    </ul>
  );
};
```

Depending on the complexity of the transformation (that is, the complexity of the code executed inside the inner function, which is passed to the `map()` method), for readability reasons, you might want to consider not using this *inline* approach (such as when mapping array elements to some complex JSX structure or when performing extra calculations as part of the mapping process). Ultimately, this comes down to personal preference and judgment.

Because it's very concise, using the `map()` method (either with the help of an extra variable or constant, or directly *inline* in the JSX code) is the de facto standard approach for outputting list data in React apps and JSX in general.

Updating Lists

Imagine you have a list of data mapped to JSX elements and a new list item is added at some point. Or, consider a scenario in which you have a list wherein two list items swap places (that is, the list is reordered). How can such updates be reflected in the DOM?

The good news is that React will take care of that for you if the update is performed in a stateful way (that is, by using React's state concept, as explained in *Chapter 4, Working with Events and State*).

However, there are a couple of important aspects to updating (stateful) lists you should be aware of.

Here's a simple example that would **not** work as intended:

```
import { useState } from 'react';

function Todos() {
  const [todos, setTodos] = useState(['Learn React', 'Recommend this book']);

  function handleAddTodo() {
    todos.push('A new todo');
  };

  return (
    <div>
      <button onClick={handleAddTodo}>Add Todo</button>
      <ul>
        {todos.map(todo => <li>{todo}</li>)}
      </ul>
    </div>
  );
};
```

Initially, two to-do items would be displayed on the screen (`<li>Learn React</li>` and `<li>Recommend this book</li>`). But once the button is clicked and `handleAddTodo` is executed, the expected result of another to-do item being displayed will not materialize.

This is because executing `todos.push('A new todo')` will update the `todos` array, but React won't notice it. Keep in mind that you must only update the state via the state updating function returned by `useState()`; otherwise, React will not re-evaluate the component function.

So, how about this code:

```
function handleAddTodo() {
  setTodos(todos.push('A new todo'));
};
```

This is also incorrect because the state updating function (`setTodos`, in this case) should receive the new state (that is, the state that should be set) as an argument. However, the `push()` method doesn't return the updated array. Instead, it mutates the existing array in place. Even if `push()` were to return the updated array, it would still be wrong to use the preceding code, because the data would be changed (mutated) behind the scenes before the state updating function would be executed. Since arrays are objects, and therefore reference data types, technically, data would be changed before informing React about that change. Following the React best practices, this should be avoided.

Therefore, when updating an array (or, as a side note, an object in general), you should perform this update in an **immutable** way (i.e., without changing the original array or object). Instead, a new array or object should be created. This new array can be based on the old array and contain all the old data, as well as any new or updated data.

Therefore, the `todos` array should be updated like this:

```
function handleAddTodo() {
  setTodos(curTodos => [...curTodos, 'A new todo']);
  // alternative: Use concat() instead of the spread operator:
  // concat(), unlike push(), returns a new array
  // setTodos(curTodos => curTodos.concat('A new todo'));
};
```

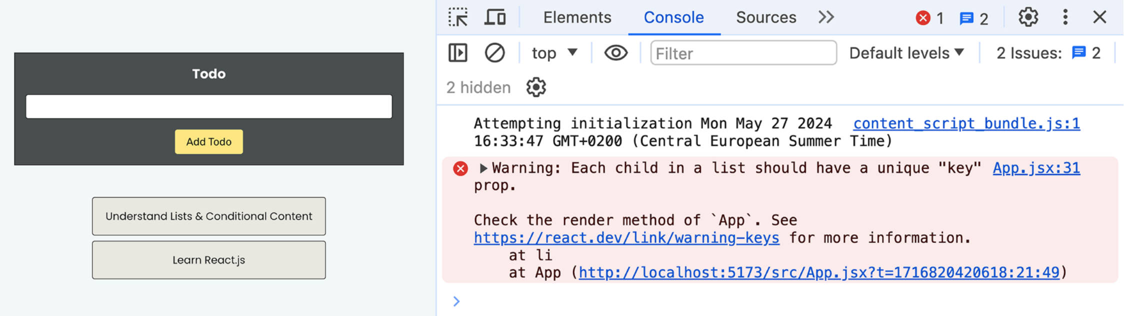
By using `concat()` or a new array, combined with the spread operator, a brand-new array is provided to the state updating function. Note also that a function is passed to the state updating function since the new state depends on the previous state.

When updating an array (or any object) state value like this, React is able to pick up those changes. Therefore, React will re-evaluate the component function and apply any required changes to the DOM.

> **Note**
>
> Immutability is not a React-specific concept, but it's a key one in React apps nonetheless. When working with state and reference values (that is, objects and arrays), immutability is extremely important to ensure that React is able to pick up changes and no "invisible" (that is, not recognized by React) state changes are performed.
>
> There are different ways of updating objects and arrays immutably, but a popular approach is to create new objects or arrays and then use the spread operator (`...`) to merge existing data into those new arrays or objects.

A Problem with List Items

If you're following along with your own code, and you output list data as described in the previous sections, you might've noticed that React actually shows a warning in the browser developer tools console, as shown in the following screenshot:



```
Attempting initialization Mon May 27 2024  content_script_bundle.js:1
16:33:47 GMT+0200 (Central European Summer Time)
▶Warning: Each child in a list should have a unique "key" App.jsx:31
prop.

Check the render method of `App`. See
https://react.dev/link/warning-keys for more information.
    at li
    at App (http://localhost:5173/src/App.jsx?t=1716820420618:21:49)
```

Figure 5.4: React sometimes generates a warning regarding missing unique keys

React is complaining about missing keys.

To understand this warning and the idea behind keys, it's helpful to explore a specific use case and a potential problem with that scenario. Assume that you have a React component that is responsible for displaying a list of items—maybe a list of to-do items. In addition, assume that those list items can be reordered and that the list can be edited in other ways (for example, new items can be added, existing items can be updated or deleted, and so on). In other words, the list is not static.

Consider this example user interface, in which a new item is added to a list of to-do items:

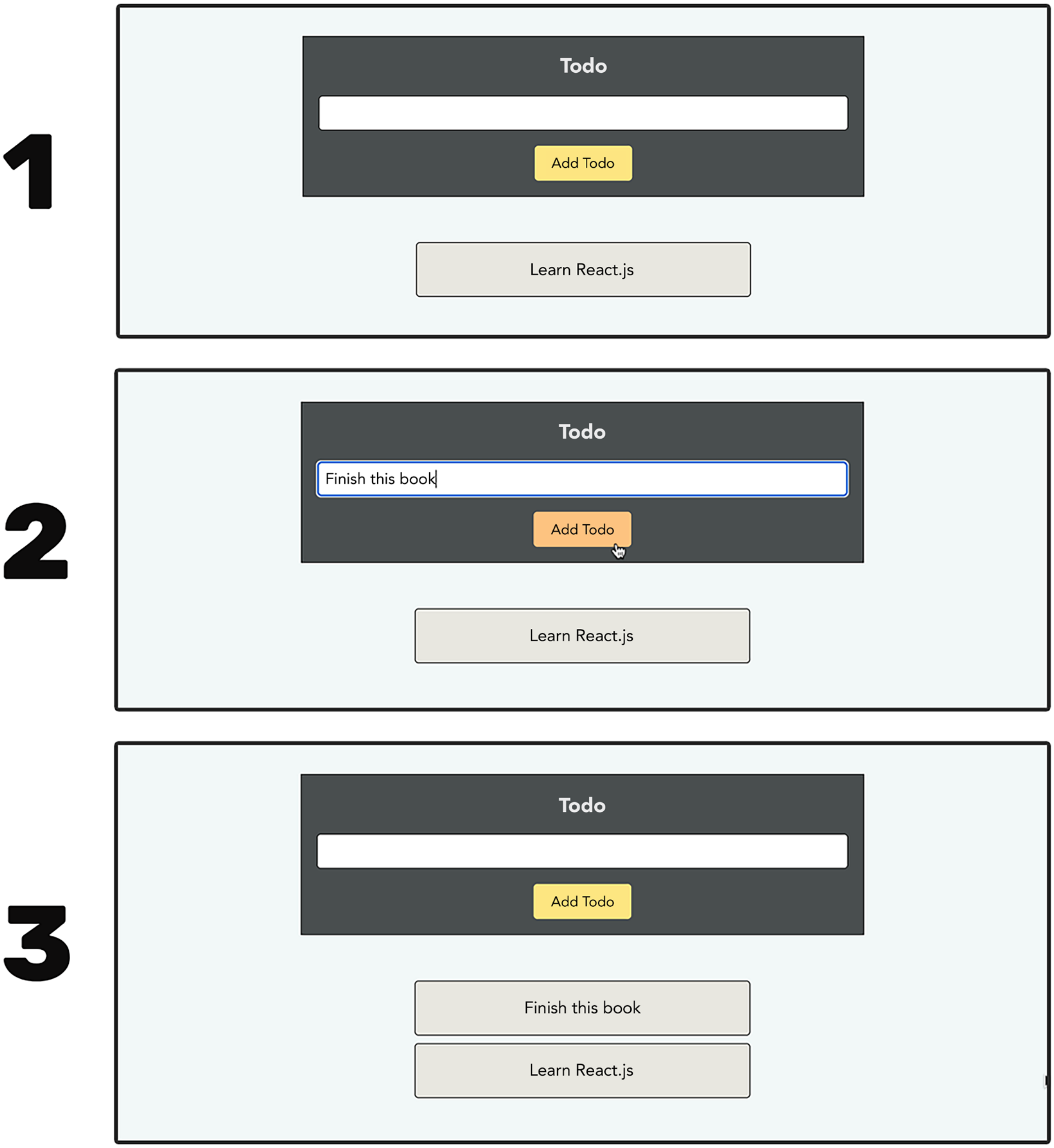

1
Todo
Add Todo
Learn React.js
2
Todo
Finish this book
Add Todo
Learn React.js
3
Todo
Add Todo
Finish this book
Learn React.js

Figure 5.5: A list gets updated by inserting a new item at the top

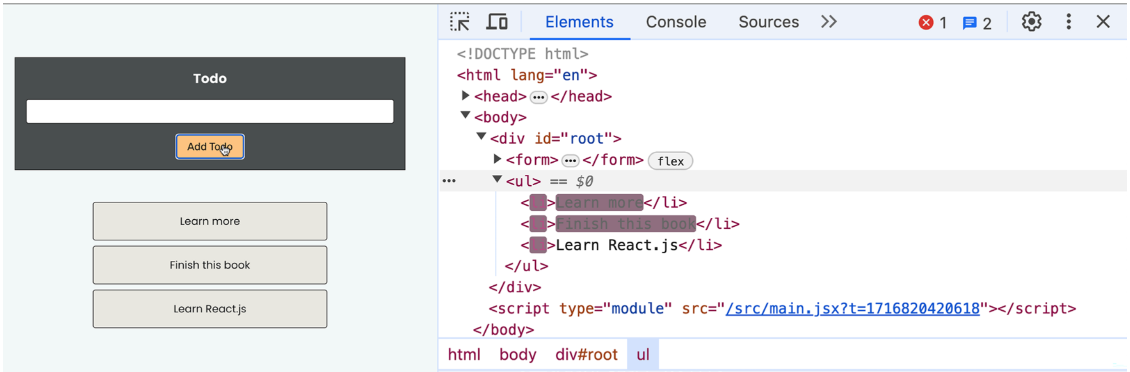
In the preceding figure, you can see the initially rendered list (*1*), which is then updated after a user enters and submits a new to-do value (*2*). A new to-do item is added to the top of the list (that is, as the first item of the list) (*3*).

Note

The example source code for this demo app can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/05-lists-conditional-code/examples/02-keys`.

If you work on this app and open the browser developer tools (and then the JavaScript console), you will see the "missing keys" warning that has been mentioned before. This app also helps with understanding where this warning is coming from.

In the Chrome DevTools, navigate to the **Elements** tab and select one of the to-do items or the empty to-do list (that is, the `<ul>` element). Once you add a new to-do item, any DOM elements that were inserted or updated are highlighted by Chrome in the **Elements** tab (by flashing briefly). Refer to the following screenshot:



```
<!DOCTYPE html>
<html lang="en">
 ▶<head> ⋯ </head>
 ▼<body>
   ▼<div id="root">
     ▶<form> ⋯ </form> flex
⋯    ▼<ul> == $0
        <li>Learn more</li>
        <li>Finish this book</li>
        <li>Learn React.js</li>
      </ul>
    </div>
    <script type="module" src="/src/main.jsx?t=1716820420618"></script>
  </body>
```

Figure 5.6: Updated DOM items are highlighted in the Chrome DevTools

The interesting part is that not only the newly added to-do element (that is, the newly inserted `<li>` element) is flashing. Instead, **all** existing `<li>` elements, which reflect existing to-do items that were not changed, are highlighted by Chrome. This implies that all these other `<li>` elements were also updated in the DOM—even though there was no need for that update. The items existed before, and their content (the to-do text) didn't change.

For some reason, React seems to destroy the existing DOM nodes (that is, the existing `<li>` items), just to then recreate them immediately. This happens for every new to-do item that is added to the list. As you might imagine, this is not very efficient and can cause performance problems for more complex apps that might be rendering dozens or hundreds of items across multiple lists.

This happens because React has no way of knowing that only one DOM node should be inserted. It cannot tell that all other DOM nodes should stay untouched because React only received a brand-new state value: a new array, filled with new JavaScript objects. Even if the content of those objects didn't change, they are technically still new objects (new values in memory).

As the developer, you know how your app works and that the content of the to-do array didn't actually change that much. But React doesn't know that. Therefore, React determines that all existing list items (`<li>` items) must be discarded and replaced by new items that reflect the new data that was provided as part of the state update. That is why **all** list-related DOM nodes are updated (that is, destroyed and recreated) for every state update.

Keys to the Rescue!

The problem outlined previously is an extremely common one. Most list updates are incremental updates, not bulk changes. But React can't tell whether that is the case for your use case and your list.

That's why React uses the concept of **keys** when working with list data and rendering list items. Keys are simply unique identifier values that can (and should) be attached to JSX elements when rendering list data. Keys help React identify elements that were rendered before and didn't change. By allowing the unique identification of all list elements, keys also help React to move (list item) DOM elements around efficiently.

Keys are added to JSX elements via the special built-in `key` prop that is accepted by every component:

```
<li key={todo.id}>{todo.text}</li>
```

This special prop can be added to all components, be they built-in or custom. You don't need to accept or handle the `key` prop in any way on your custom components; React will do that for you automatically.

The `key` prop requires a value that is unique for every list item. No two list items should have the same key. In addition, good keys are directly attached to the underlying data that makes up the list item. Therefore, list item indexes are poor keys because the index isn't attached to the list item data. If you reorder items in a list, the indexes stay the same (an array always starts with index `0`, followed by `1`, and so on) but the data is changed.

Consider the following example:

```
const hobbies = ['Sports', 'Cooking'];
const reversed = hobbies.reverse(); // ['Cooking', 'Sports']
```

In this example, `'Sports'` has the index `0` in the `hobbies` array. In the `reversed` array, its index would be `1` (because it's the second item now). In this case, if the index were used as a key, the data would not be attached to it.

Good keys are unique `id` values, such that every `id` belongs to exactly one value. If that value moves or is removed, its `id` should move or disappear with that value.

Finding good `id` values typically isn't a huge problem since most list data is fetched from databases anyway. No matter whether you're dealing with products, orders, users, or shopping cart items, it's all data that would typically be stored in a database. This kind of data already has unique `id` values since you always have some kind of unique identification criteria when storing data in databases.

Sometimes, even the values themselves can be used as keys. Consider the following example:

```
const hobbies = ['Sports', 'Cooking'];
```

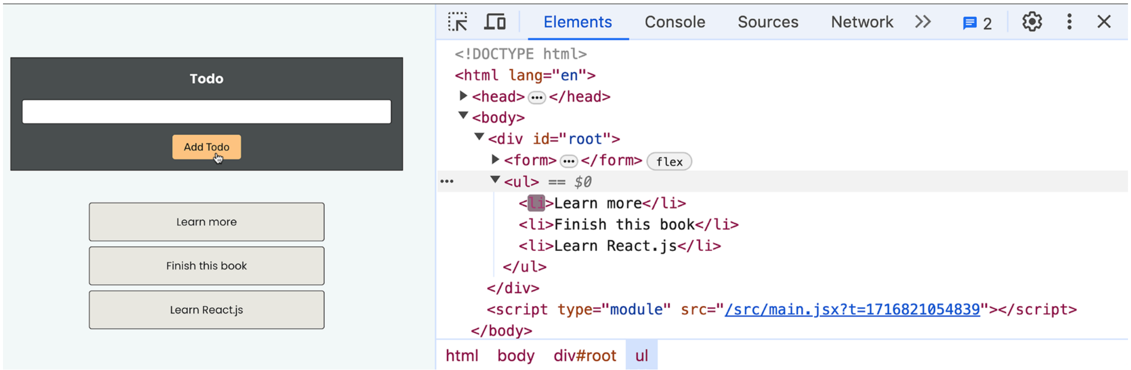
Hobbies are `string` values, and there is no unique `id` value attached to individual hobbies. Every hobby is a primitive value (a `string`). However, in cases like this, you typically won't have duplicate values as it doesn't make sense for a hobby to be listed more than once in an array like this. Therefore, the values themselves qualify as good keys:

```
hobbies.map(hobby => <li key={hobby}>{hobby}</li>);
```

In cases where you can't use the values themselves and there is no other possible key value, you can generate unique `id` values directly in your React app code. As a last resort, you can also fall back to using indexes; but be aware that this can lead to unexpected bugs and side effects if you reorder list items.

With keys added to list item elements, React is able to identify all items correctly. When the component state changes, it can identify JSX elements that were rendered before already. Those elements are therefore not destroyed or recreated anymore.

You can confirm this by again opening the browser DevTools to check which DOM elements are updated upon changes to the underlying list data:



```
<!DOCTYPE html>
<html lang="en">
 ▶<head> ··· </head>
 ▼<body>
   ▼<div id="root">
     ▶<form> ··· </form> flex
··· ▼<ul> == $0
        <li>Learn more</li>
        <li>Finish this book</li>
        <li>Learn React.js</li>
      </ul>
    </div>
    <script type="module" src="/src/main.jsx?t=1716821054839"></script>
  </body>
```

Figure 5.7: From multiple list items, only one DOM element gets updated

After adding keys, when updating the list state, only the new DOM item is highlighted in the Chrome DevTools. The other items are (correctly) ignored by React.

Summary and Key Takeaways

- Like any other JavaScript value, JSX elements can be set and changed dynamically, based on different conditions.
- Content can be set conditionally via `if` statements, ternary expressions, the logical "and" operator (&&), or in any other way that works in JavaScript.
- There are multiple ways to handle conditional content—any approach that would work in vanilla JavaScript can also be used in React apps.
- Arrays with JSX elements can be inserted into JSX code and will lead to the array elements being outputted as sibling DOM elements.

- List data can be converted into JSX element arrays via `for` loops, the `map()` method, or any other JavaScript approach that leads to a similar conversion.
- Using the `map()` method is the most common way of converting list data to JSX element lists.
- Keys (via the `key` prop) should be added to the list JSX elements to help React update the DOM efficiently.

What's Next?

With conditional content and lists, you now have all the key tools needed to build both simple and more complex user interfaces with React. You can hide and show elements or groups of elements as needed, and you can dynamically render and update lists of elements to output lists of products, orders, or users.

Of course, that's not all that's needed to build realistic user interfaces. Adding logic for changing content dynamically is one thing, but most web apps also need CSS styling that should be applied to various DOM elements. This book is not about CSS, but the next chapter will still explore how React apps can be styled. Especially when it comes to setting and changing styles dynamically or scoping styles to specific components, there are various React-specific concepts that should be familiar to every React developer.

Test Your Knowledge!

Test your knowledge about the concepts covered in this chapter by answering the following questions. You can then compare your answers to examples that can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/05-lists-conditional-code/exercises/questions-answers.md`:

1. What is "conditional content"?
2. Name at least two different ways of rendering JSX elements conditionally.
3. Which elegant approach can be used to define element tags conditionally?
4. What's a potential downside of using only ternary expressions (for conditional content)?
5. How can lists of data be rendered as JSX elements?
6. Why should keys be added to rendered list items?
7. Give one example each for a good and a bad key.

Apply What You Learned

You are now able to use your React knowledge to change dynamic user interfaces in a variety of ways. Besides being able to change displayed text values and numbers, you can now also hide or show entire elements (or chunks of elements) and display lists of data.

In the following sections, you will find two activities that allow you to apply your newly gained knowledge (combined with the knowledge gained in the other book chapters).

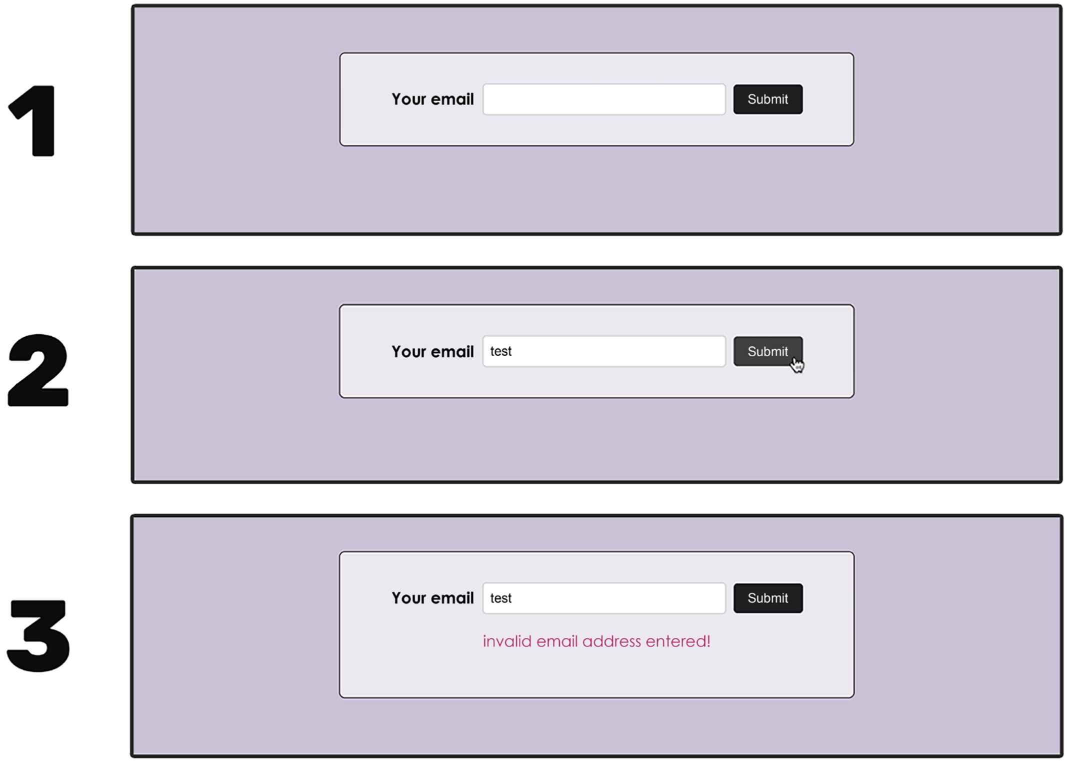
Activity 5.1: Showing a Conditional Error Message

In this activity, you'll build a basic form that allows users to enter their email address. Upon form submission, the user input should be validated and invalid email addresses (for simplicity, here email addresses that contain no @ sign are being referred to) should lead to an error message being shown below the form. When invalid email addresses are made valid, potentially visible error messages should be removed again.

Perform the following steps to complete this activity:

1. Build a user interface that contains a form with a label, an input field (of the text type—to make entering incorrect email addresses easier for demo purposes), and a submit button that leads to the form being submitted.
2. Collect the entered email address and show an error message below the form if the email address contains no @ sign.

The final user interface should look and work as shown here:



invalid email address entered!

Figure 5.8: The final user interface of this activity

Note

Styling will, of course, differ. To get the same styling as shown in the screenshot, use my prepared starting project, which you can find here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/05-lists-conditional-code/activities/practice-1-start`.

Analyze the `index.css` file in that project to determine how to structure your JSX code to apply the styles.

Note

You can find the full example solution here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/05-lists-conditional-code/activities/practice-1`.

Activity 5.2: Outputting a List of Products

In this activity, you will build a user interface where a list of (dummy) products is displayed on the screen. The interface should also contain a button that, when clicked, adds another new (dummy) item to the existing list of products.

Perform the following steps to complete this activity:

1. Add a list of dummy product objects (every object should have an ID, title, and price) to a React component and add code to output these product items as JSX elements.
2. Add a button to the user interface. When clicked, the button should add a new product object to the product data list. This should then cause the user interface to update and display an updated list of product elements.

The final user interface should look and work as shown here:

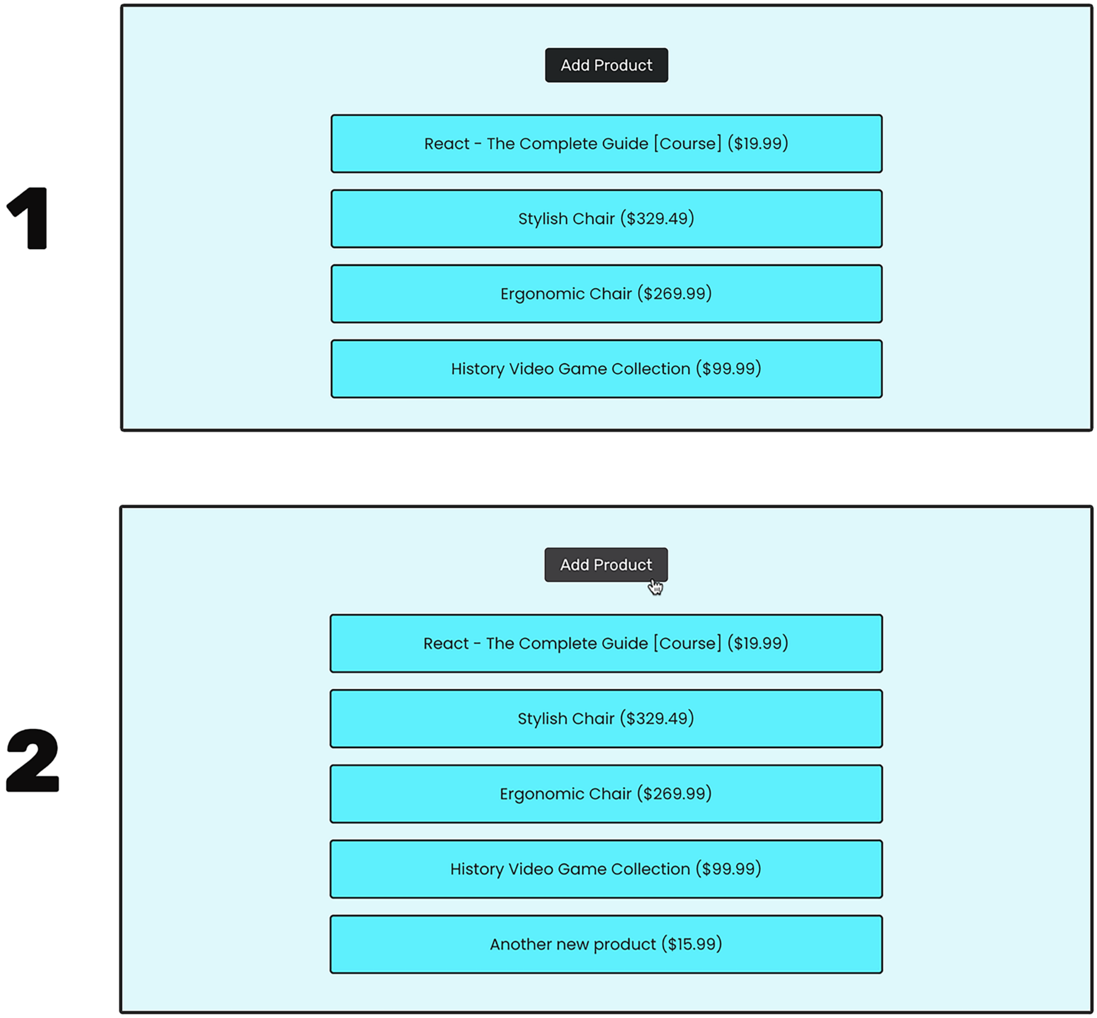

1
Add Product
React - The Complete Guide [Course] ($19.99)
Stylish Chair ($329.49)
Ergonomic Chair ($269.99)
History Video Game Collection ($99.99)
2
Add Product
React - The Complete Guide [Course] ($19.99)
Stylish Chair ($329.49)
Ergonomic Chair ($269.99)
History Video Game Collection ($99.99)
Another new product ($15.99)

Figure 5.9: The final user interface of this activity

Note

Styling will, of course, differ. To get the same styling as shown in the screenshot, use my prepared starting project, which you can find here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/05-lists-conditional-code/activities/practice-2-start`.

Analyze the `index.css` file in that project to determine how to structure your JSX code to apply the styles.

Note

You can find the full example solution here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/05-lists-conditional-code/activities/practice-2`.

6

Styling React Apps

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Style JSX elements via inline style assignments or with the help of CSS classes
- Set inline and class styles, both statically and dynamically or conditionally
- Build reusable components that allow for style customization
- Utilize CSS Modules to scope styles to components
- Understand the core idea behind `styled-components`, a third-party CSS-in-JS library
- Use Tailwind CSS to style React apps

Introduction

React.js is a frontend JavaScript library. This means that it's all about building (web) user interfaces and handling user interaction.

Up to this point, this book has extensively explored how React can be used to add interactivity to a web app. State, event handling, and dynamic content are key concepts relating to this.

Of course, websites and web apps are not just about interactivity. You could build an amazing web app that offers interactive and engaging features, and yet it may still be unpopular if it lacks appealing visuals. Presentation is key, and the web is no exception.

Therefore, like all other apps and websites, React apps and websites need proper styling, and when working with web technologies, **Cascading Style Sheets** (**CSS**) is the language of choice.

However, this book is not about CSS. It won't explain or teach you how to use CSS, as there are dedicated, better resources for that (e.g., the free CSS guides at `https://developer.mozilla.org/en-US/docs/Learn/CSS`). But this chapter will teach you how to combine CSS code with JSX and React concepts, such as state and props. You will learn how to add styles to your JSX elements, style custom components, and make those components' styles configurable. This chapter will also teach you how to set styles dynamically and conditionally and explore popular third-party libraries, like styled-components and Tailwind CSS, that can be used for styling.

How Does Styling Work in React Apps?

Up to this point, the apps and examples presented in this book have only had minimal styling. But they at least had some basic styling, rather than no styling at all.

But how was that styling added? How can styles be added to user interface elements (such as DOM elements) when using React?

The short answer is, "Just as you would to non-React apps." You can add CSS styles and classes to JSX elements just as you would to regular HTML elements. And in your CSS code, you can use all the features and selectors you know from CSS. There are no React-specific changes you have to make when writing CSS code.

The code examples used so far (i.e., the activities or other examples hosted on GitHub) always used regular CSS styling, with the help of CSS selectors, to apply some basic styles to the final user interface. Those CSS rules were defined in an `index.css` file, which is part of every newly created React project (when using Vite for project creation, as shown in *Chapter 1*, *React – What and Why*).

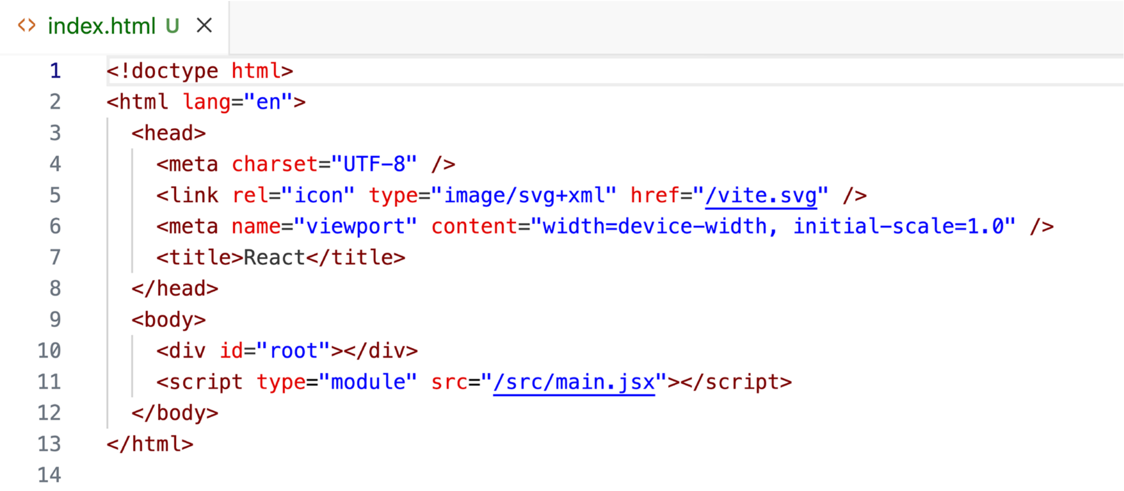
For example, here's the `index.css` file used in *Activity 5.2* of the previous chapter (*Chapter 5*, *Rendering Lists and Conditional Content*):

```css
@import url('https://fonts.googleapis.com/css2?family=Poppins:wght@400;700&family=Rubik:ital,wght@0,300..900;1,300..900&display=swap');

body {
  margin: 0;
  padding: 3rem;
  font-family: 'Poppins', sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  background-color: #dff8fb;
  color: #212324;
}

button {
  padding: 0.5rem 1rem;
  font-family: 'Rubik', sans-serif;
  font-size: 1rem;
  border: none;
  border-radius: 4px;
  background-color: #212324;
  color: #fff;
  cursor: pointer;
}

button:hover {
  background-color: #3f3e40;
}

ul {
  max-width: 35rem;
  list-style-type: none;
  padding: 0;
  margin: 2rem auto;
}

li {
  margin: 1rem 0;
  padding: 1rem;
  background-color: #5ef0fd;
  border: 2px solid #212324;
  border-radius: 4px;
}
```

The actual CSS code and its meaning are not important (as mentioned, this book is not about CSS). However, what is important is the fact that this code contains no JavaScript or React code at all. As mentioned, the CSS code you write is totally independent of the fact that you're using React in your app.

The more interesting question is, how is that code actually applied to the rendered web page? How is it imported into that page?

Normally, you would expect style file imports (via `<link href="…">`) inside of the HTML files that are served. Since React apps are typically about building **single-page applications** (see *Chapter 1*, *React – What and Why*), you only have one HTML file—the `index.html` file. But if you inspect that file, you won't find any `<link href="…">` import that would point to the `index.css` file (only some other `<link>` element that imports a favicon), as you can see in the following screenshot:



```html
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>React</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

Figure 6.1: The <head> section of the index.html file contains no <link> import that points to the index.css file

How are the styles defined in `index.css` imported and applied then?

You find an `import` statement in the root entry file (this is the `main.jsx` file in projects generated via `Vite`):

```jsx
import React from 'react';
import ReactDOM from 'react-dom/client';

import App from './App.jsx';
import './index.css';

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
);
```

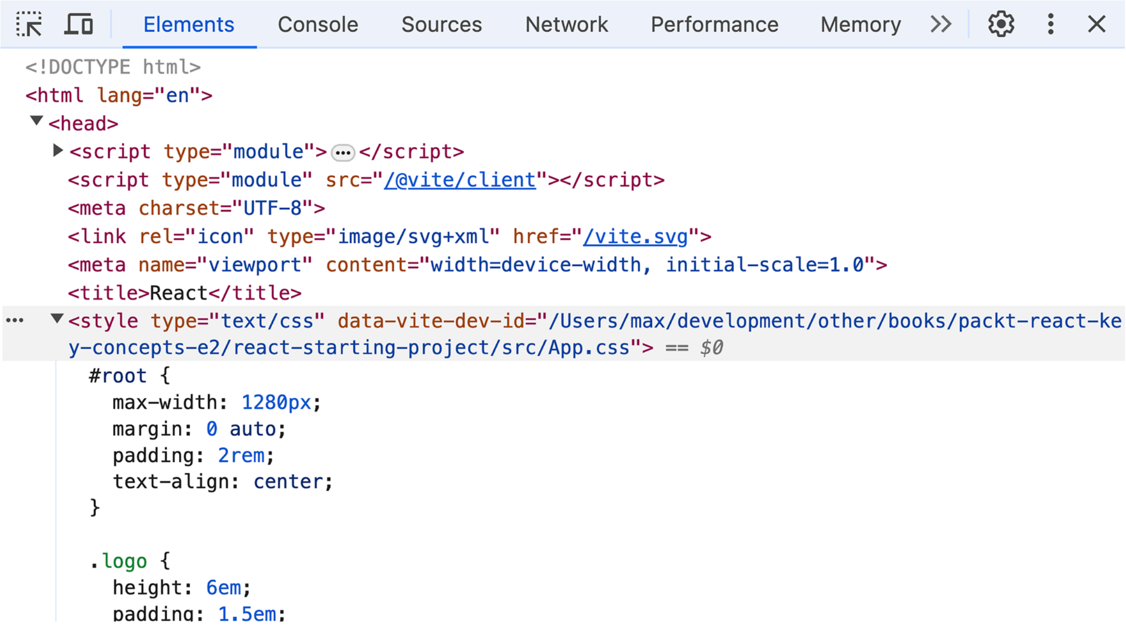
The `import './index.css';` statement leads to the CSS file being imported and the defined CSS code being applied to the rendered web page.

It is worth noting that this is not standard JavaScript behavior. You can't import CSS files into JavaScript—at least, not if you're just using vanilla JavaScript.

CSS works this way in React apps because the code is transpiled before it's loaded into the browser. Therefore, you won't find that `import` statement in the final JavaScript code that's executed in the browser. Instead, during the **transpilation process**, the transpiler identifies the CSS import, removes it from the JavaScript file, and injects the CSS code (or an appropriate link to the potentially bundled and optimized CSS file) into the `index.html` file.

You can confirm this by inspecting the rendered **Document Object Model** (**DOM**) content of the loaded web page in the browser.

To do so, select the **Elements** tab in developer tools in Chrome, as shown below:



```
<!DOCTYPE html>
<html lang="en">
▼<head>
  ▶<script type="module">⋯</script>
   <script type="module" src="/@vite/client"></script>
   <meta charset="UTF-8">
   <link rel="icon" type="image/svg+xml" href="/vite.svg">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>React</title>
••• ▼<style type="text/css" data-vite-dev-id="/Users/max/development/other/books/packt-react-ke
   y-concepts-e2/react-starting-project/src/App.css"> == $0
     #root {
       max-width: 1280px;
       margin: 0 auto;
       padding: 2rem;
       text-align: center;
     }

     .logo {
       height: 6em;
       padding: 1.5em;
```

Figure 6.2: Injected CSS <style> elements can be found in the DOM at runtime

You can define any styles you want to apply to your HTML elements (that is, to your JSX elements in your components) directly inside of the `index.css` file, or in any other CSS files that are imported by the `index.css` file.

You could also add additional CSS import statements, pointing at other CSS files, to the `main.jsx` file or any other JavaScript files (including files that store components). However, it is important to keep in mind that CSS styles are always global. No matter whether you import a CSS file into `main.jsx` or a component-specific JavaScript file, the styles defined in that CSS file will be applied globally.

That means that styles defined in a `goal-list.css` file, which may be imported in a `GoalList.jsx` file, could still affect JSX elements defined in a totally different component. Later in this chapter, you will learn about techniques that allow you to prevent accidental style clashes and implement style scoping.

Using Inline Styles

You can use CSS files to define global CSS styles and use different CSS selectors to target different JSX elements (or groups of elements).

But even though it's typically discouraged, you can also set inline styles directly on JSX elements via the `style` prop.

Note

If you're wondering why inline styles are discouraged, the following discussion on Stack Overflow provides many arguments against inline styles: `https://stackoverflow.com/questions/2612483/whats-so-bad-about-in-line-css`.

Setting inline styles in JSX code works like this:

```
function TodoItem() {
  return <li style={{color: 'red', fontSize: '18px'}}>Learn React!</li>;
};
```

In this example, the `style` prop is added to the `<li>` element (all JSX elements support the `style` prop), and both the `color` and `size` properties of the text are set via CSS.

This approach differs from what you would use to set inline styles when working with just HTML (instead of JSX). When using plain HTML, you would set inline styles like this:

```
<li style="color: red; font-size: 18px">Learn React!</li>
```

The difference is that the `style` prop expects to receive a JavaScript object that contains the style settings—not a plain string. This is something that must be kept in mind, since, as mentioned previously, inline styles typically aren't used that often.

Since the `style` object is an object and not a plain string, it is passed as a value between curly braces—just as an array, a number, or any other non-string value would have to be set between curly braces (anything between double or single quotes is treated as a string value). Therefore, it's worth noting that the preceding example does not use any kind of special "double curly-braces" syntax and, instead, uses one pair of curly braces to surround the non-string value and another pair to surround the object data.

Inside the `style` object, any CSS style properties supported by the underlying DOM element can be set. The property names are those defined for the HTML element (i.e., the same CSS property names you could target and set with vanilla JavaScript, when mutating an HTML element).

When setting styles in JavaScript code (as with the `style` prop shown above), JavaScript CSS property names have to be used. Those names are similar to the CSS property names you would use in CSS code but not quite the same. Differences occur for property names that consist of multiple words (e.g., `font-size`). When targeting such properties in JavaScript, camelCase notation must be used (`fontSize` instead of `font-size`), as JavaScript properties cannot contain dashes. Alternatively, you could wrap the property name with quotes (`'font-size'`).

Note

You can find more information about the HTML element style property and JavaScript CSS property names here: `https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/style`.

Setting Styles via CSS Classes

As mentioned, using inline styles is typically discouraged, and therefore, CSS styles defined in CSS files (or between `<style>` tags in the document `<head>` section) are preferred.

In those CSS code blocks, you can write regular CSS code and use CSS selectors to apply CSS styles to certain elements. You could, for example, style all `<li>` elements on a page (no matter which component may have rendered them) like this:

```
li {
  color: red;
  font-size: 18px;
}
```

As long as this code gets added to the page (because the CSS file in which it is defined is imported into `main.jsx`, for instance), the styling will be applied.

Quite frequently, developers aim to target specific elements or groups of elements. Instead of applying some style to all `<li>` elements on a page, the goal could be to only target the `<li>` elements that are part of a specific list. Consider this HTML structure that's rendered to the page (it may be split across multiple components, but this doesn't matter here):

```
<nav>
  <ul>
    <li><a href="…">Home</a></li>
    <li><a href="…">New Goals</a></li>
  </ul>
</nav>
...
<h2>My Course Goals</h2>
<ul>
  <li>Learn React!</li>
  <li>Master React!</li>
</ul>
```

In this example, the navigation list items will most likely not receive the same styling as the `course goal` list items (and vice versa).

Typically, this problem would be solved with the help of CSS classes and the class selector. You could adjust the HTML code like this:

```
<nav>
  <ul>
    <li><a href="…">Home</a></li>
    <li><a href="…">New Goals</a></li>
  </ul>
```

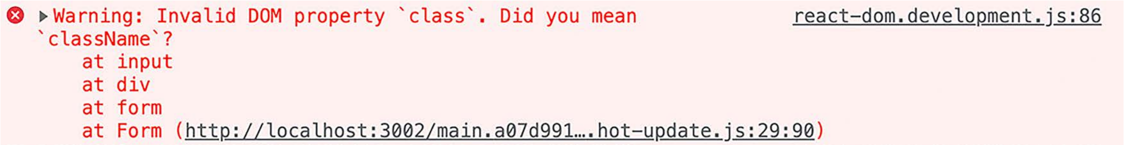
```
</nav>
...
<h2>My Course Goals</h2>
<ul>
  <li class="goal-item">Learn React!</li>
  <li class="goal-item">Master React!</li>
</ul>
```

The following CSS code would then only target the course goal list items but not the navigation list items:

```
.goal-item {
  color: red;
  font-size: 18px;
}
```

This approach almost works in React apps as well.

However, if you try to add CSS classes to JSX elements, as shown in the previous example, you will face a warning in the browser's developer tools:



```
▶ Warning: Invalid DOM property `class`. Did you mean          react-dom.development.js:86
`className`?
    at input
    at div
    at form
    at Form (http://localhost:3002/main.a07d991….hot-update.js:29:90)
```

Figure 6.3: A warning output by React

As illustrated in the preceding figure, you should not add `class` as a prop and, instead, use `className`. Indeed, if you swap `class` for `className` as a prop name, the warning will disappear, and the class CSS styles will be applied. Hence, the proper JSX code looks like this:

```
<ul>
  <li className="goal-item">Learn React!</li>
  <li className="goal-item">Master React!</li>
</ul>
```

But why is React suggesting you use `className` instead of `class`?

It's similar to using `htmlFor` instead of `for` when working with `<label>` objects (as discussed in *Chapter 4*, *Working with Events and State*). Just like `for`, `class` is a keyword in JavaScript, and therefore, `className` is used as a prop name instead.

Setting Styles Dynamically

With inline styles and CSS classes (and global CSS styles in general), there are various ways of applying styles to elements. Thus far, all examples have shown static styles—that is, styles that will never change once a page has been loaded.

But while most page elements don't change their styles after a page is loaded, you also typically have some elements that should be styled dynamically or conditionally. Here are some examples:

- A to-do app where different to-do priorities receive different colors
- An input form where invalid form elements should be highlighted following an unsuccessful form submission
- A web-based game where players can choose colors for their avatars

In such cases, applying static styles is not enough, and dynamic styles should be used instead. Setting styles dynamically is straightforward. Again, it's just about applying the key React concepts covered earlier (most importantly, those regarding the setting of dynamic values from *Chapter 2*, *Understanding React Components and JSX*, and *Chapter 4*, *Working with Events and State*).

Here's an example where the color of a paragraph is set dynamically to the color that a user enters into an input field:

```
function ColoredText() {
  const [enteredColor, setEnteredColor] = useState('');

  function handleUpdateTextColor(event) {
    setEnteredColor(event.target.value);
  };

  return (
    <>
      <input type="text" onChange={handleUpdateTextColor}/>
      <p style={{color: enteredColor}}>This text's color changes dynamically!</
p>
    </>
  );
};
```

The text entered in the `<input>` field is stored in the `enteredColor` state. This state is then used to set the `color` CSS property of the `<p>` element dynamically. This is achieved by passing a `style` object, with the `color` property set to the `enteredColor` value as a value to the `style` prop of the `<p>` element. The text color of the paragraph is, therefore, set dynamically to the value entered by the user (assuming that users enter valid CSS color values into the `<input>` field).

You're not limited to inline styles; CSS classes can also be set dynamically, as in the following snippet:

```
function TodoPriority() {
  const [chosenPriority, setChosenPriority] = useState('low-prio');

  function handleChoosePriority(event) {
    setChosenPriority(event.target.value);
  };

  return (
    <>
      <p className={chosenPriority}>Chosen Priority: {chosenPriority}</p>
      <select onChange={handleChoosePriority}>
        <option value="low-prio">Low</option>
        <option value="high-prio">High</option>
      </select>
    </>
  );
};
```

In this example, the `chosenPriority` state will alternate between `low-prio` and `high-prio`, depending on the drop-down selection. The state value is then output as text inside the paragraph and is also used as a dynamic CSS class name, applied to the `<p>` element. For this to have any visual effect, there must, of course, be `low-prio` and `high-prio` CSS classes defined in some CSS file or `<style>` block. For example, consider the following code in `index.css`:

```
.low-prio {
  background-color: blue;
  color: white;
}

.high-prio {
  background-color: red;
  color: white;
}
```

Conditional Styles

Closely related to **dynamic styles** are **conditional styles**. In fact, ultimately, they are really just a special case of dynamic styles. In the previous examples, inline style values and class names were set as equal to values chosen or entered by the user.

However, you can also derive styles or class names dynamically based on different conditions, as shown here:

```
function TextInput({isValid, isRecommended, ...props}) {
  let cssClass = 'input-default';

  if (isRecommended) {
    cssClass = 'input-recommended';
  }

  if (!isValid) {
    cssClass = 'input-invalid';
  }

  return <input className={cssClass} {...props} />
};
```

In this example, a wrapper component around the standard `<input>` element is built. (For more information about wrapper components, see *Chapter 3*, *Components and Props*.) The main purpose of this wrapper component is to set some default styles for the wrapped `<input>` element. The **wrapper component** is built to provide a pre-styled input element that can be used anywhere in the app. Indeed, providing pre-styled elements is one of the most common and popular use cases for building wrapper components.

In this concrete example, the default styles are applied using CSS classes. If the `isValid` prop value is `true` and the value of the `isRecommended` prop is `false`, the `input-default` CSS class will be applied to the `<input>` element, since neither of the two `if` statements become active.

If `isRecommended` is `true` (but `isValid` is `false`), the `input-recommended` CSS class would be applied. If `isValid` is `false`, the `input-invalid` class is added instead. Of course, the CSS classes must be defined in some imported CSS files (for example, in `index.css`).

Inline styles could be set in a similar way, as shown in the following snippet:

```
function TextInput({isValid, isRecommended, ...props}) {
  let bgColor = 'black';

  if (isRecommended) {
    bgColor = 'blue';
  }

  if (!isValid) {
    bgColor = 'red';
  }

  return <input style={{backgroundColor: bgColor}} {...props} />
};
```

In this example, the background color of the `<input>` element is set conditionally, based on the values received via the `isValid` and `isRecommended` props.

Combining Multiple Dynamic CSS Classes

In previous examples, a maximum of one CSS class was set dynamically at a time. However, it's not uncommon to encounter scenarios where multiple dynamically derived CSS classes should be merged and added to an element.

Consider the following example:

```
function ExplanationText({children, isImportant}) {
  const defaultClasses = 'text-default text-expl';

  return <p className={defaultClasses}>{children}</p>;
}
```

Here, two CSS classes are added to `<p>` by simply combining them into one string. Alternatively, you could directly add a string with the two classes like this:

```
return <p className="text-default text-expl">{children}</p>;
```

This code will work, but what if the goal is to also add another class name to the list of classes, based on the `isImportant` prop value (which is ignored in the preceding example)?

Replacing the default list of classes is easy, as you have learned:

```
function ExplanationText({children, isImportant}) {
  let cssClasses = 'text-default text-expl';

  if (isImportant) {
    cssClasses = 'text-important';
  }

  return <p className={cssClasses}>{children}</p>;
}
```

But what if the goal is not to replace the list of default classes? What if `text-important` should be added as a class to `<p>`, in addition to `text-default` and `text-expl`?

The `className` prop expects to receive a string value, so passing an array of classes isn't an option. However, you can simply merge multiple classes into one string, and there are a couple of different ways to do that:

- String concatenation:

  ```
  cssClasses = cssClasses + ' text-important';
  ```

- Using a template literal:

  ```
  cssClasses = `${cssClasses} text-important`;
  ```

- Joining an array:

  ```
  cssClasses = [cssClasses, 'text-important'].join(' ');
  ```

These examples could all be used inside the `if` statement (`if (isImportant)`) to conditionally add the `text-important` class, based on the `isImportant` prop value. All three approaches, as well as variations of these approaches, will work because all these approaches produce a string. In general, any approach that yields a string can be used to generate values for `className`.

Merging Multiple Inline Style Objects

When working with inline styles, instead of CSS classes, you can also merge multiple style objects. The main difference is that you don't produce a string with all values but, rather, an object with all combined style values.

This can be achieved by using standard JavaScript techniques to merge multiple objects into one object. The most popular technique involves using the **spread operator**, as shown in this example:

```
function ExplanationText({children, isImportant}) {
  let defaultStyle = { color: 'black' };

  if (isImportant) {
    defaultStyle = { ...defaultStyle, backgroundColor: 'red' };
  }

  return <p style={defaultStyle}>{children}</p>;
}
```

Here, you will observe that `defaultStyle` is an object with a `color` property. If `isImportant` is `true`, it's replaced with an object that contains all the properties it had before (via the spread operator, `...defaultStyle`) as well as the `backgroundColor` property.

Note

For more information on the function and use of the spread operator, see *Chapter 5*, *Rendering Lists and Conditional Content*.

Building Components with Customizable Styles

As you are aware by now, components can be reused. This is supported by the fact that they can be configured via props. The same component can be used in different places on a page with different configurations to yield a different output.

Since styles can be set both statically and dynamically, you can also make the styling of your components customizable. The preceding examples already show such customization in action; for example, the `isImportant` prop was used in the previous example to conditionally add a `red` background color to a paragraph. The `ExplanationText` component therefore already allows indirect style customization via the `isImportant` prop.

Besides this form of customization, you could also build components that accept props already holding CSS class names or style objects. For example, the following wrapper component accepts a `className` prop that is merged with a default CSS class (`btn`):

```
function Button({children, config, className}) {
  return <button {...config} className={`btn ${className}`}>{children}</but-
ton>;
};
```

This component could be used in another component in the following way:

```
<Button config={{onClick: doSomething}} className="btn-alert">Click me!</But-
ton>
```

If used like this, the final `<button>` element would receive both the `btn` as well as `btn-alert` classes.

You don't have to use `className` as a prop name; any name can be used, since it's your component. However, it's not a bad idea to use `className` because you can then keep your mental model of setting CSS classes via `className` (for built-in components, you will not have that choice).

Instead of merging prop values with default CSS class names or style objects, you can overwrite default values. This allows you to build components that come with some styling out of the box without enforcing that styling:

```
function Button({children, config, className}) {
  let cssClasses = 'btn';
  if (className) {
    cssClasses = className;
  }
  return <button {...config} className={cssClasses}>{children}</button>;
};
```

You can see how all the different concepts covered throughout this book are coming together here: props allow customization, values can be set, swapped, and changed dynamically and conditionally, and therefore, highly reusable and configurable components can be built.

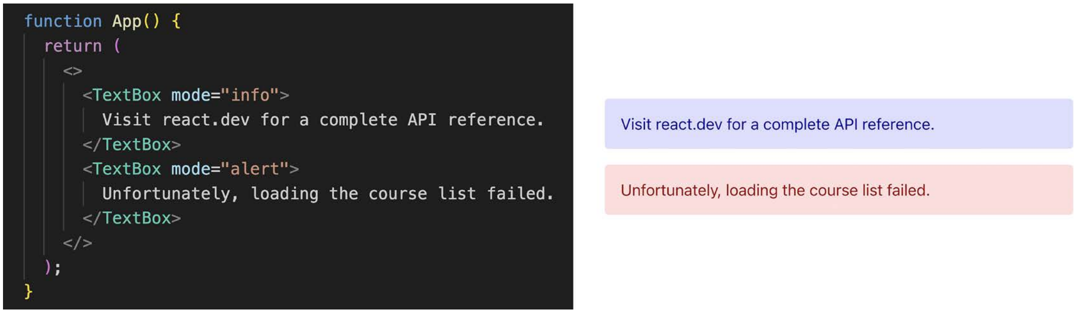
Customization with Fixed Configuration Options

Besides exposing props such as `className` or `style`, which are merged with other classes or styles defined inside a component function, you can also build components that apply different styles or class names based on other prop values.

This has been shown in the previous examples where props such as `isValid` or `isImportant` were used to apply certain styles conditionally. This way of applying styles could, therefore, be called "indirect styling" (although this is not an official term).

Both approaches can shine in different circumstances. For wrapper components, for example, accepting `className` or `style` props (which can be merged with other styles inside the component) enables the component to be used just like a built-in component (e.g., like the component it wraps). Indirect styling, on the other hand, can be very useful if you want to build components that provide a couple of pre-defined variations.

A good example is a text box that provides two built-in themes that can be selected via a specific prop.



```
function App() {
  return (
    <>
      <TextBox mode="info">
        Visit react.dev for a complete API reference.
      </TextBox>
      <TextBox mode="alert">
        Unfortunately, loading the course list failed.
      </TextBox>
    </>
  );
}
```

Visit react.dev for a complete API reference.

Unfortunately, loading the course list failed.

Figure 6.4: A TextBox is styled based on the value of the "mode" prop

The code for the `TextBox` component could look like this:

```
function TextBox({children, mode}) {
  let cssClasses;

  if (mode === 'alert') {
    cssClasses = 'box-alert';
  } else if (mode === 'info') {
    cssClasses = 'box-info';
  }

  return <p className={cssClasses}>{children}</p>;
};
```

This `TextBox` component always yields a paragraph element. If the `mode` prop is set to any value other than `'alert'` or `'info'`, the paragraph doesn't receive any special styling. But if `mode` is equal to `'alert'` or `'info'`, specific CSS classes are added to the paragraph.

This component, therefore, doesn't allow direct styling via some `className` or `style` prop that would be merged, but it does offer different variations or themes that can be set with the help of a specific prop (the `mode` prop in this case).

The Problem with Unscoped Styles

If you consider the different examples you've so far dealt with in this chapter, there's one specific use case that occurs quite frequently: styles are relevant to a specific component only.

For example, in the `TextBox` component in the previous section, `'box-alert'` and `'box-info'` are CSS classes that are likely only relevant for this specific component and its markup. If any other JSX element in the app had a `'box-alert'` class applied to it (even though that might be unlikely), it probably shouldn't be styled the same as the `<p>` element in the `TextBox` component.

Styles from different components could clash with each other and overwrite each other because styles are not scoped (i.e., restricted) to a specific component. CSS styles are always global, unless inline styles are used (which is discouraged, as mentioned earlier).

When working with component-based libraries such as React, this lack of scoping is a common issue. It's easy to write conflicting styles as app sizes and complexities grow (or, in other words, as more and more components are added to the code base of a React app).

That's why various solutions for this problem have been developed by members of the React community. The following are three of the most popular solutions:

- CSS Modules (supported out of the box in React projects created with Vite)
- Styled components (using a third-party library called `styled-components`)
- Tailwind CSS (a popular CSS library)

Scoped Styles with CSS Modules

CSS Modules is the name for an approach where individual CSS files are linked to specific JavaScript files and the components defined in those files. This link is established by transforming CSS class names, such that every JavaScript file receives its own, unique CSS class names. This transformation is performed automatically as part of the code build workflow. Therefore, a given project setup must support CSS Modules by performing the described CSS class name transformation. Projects created via Vite support CSS Modules by default.

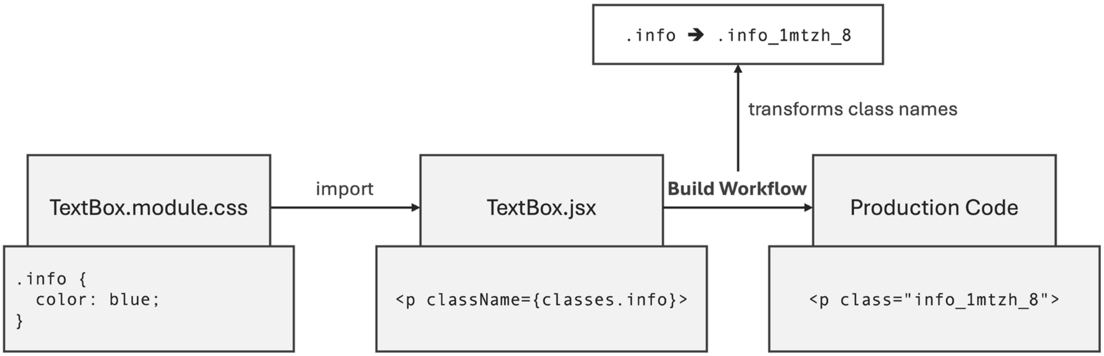


Figure 6.5: CSS modules in action. CSS class names are transformed into unique names during the build workflow

CSS Modules are enabled and used by naming CSS files in a very specific and clearly defined way: `<anything>.module.css`. `<anything>` is any value of your choosing, but the `.module` part in front of the file extension is required, as it signals (to the project build workflow) that this CSS file should be transformed according to the CSS Modules approach.

Therefore, CSS files named like this must be imported into components in a specific way:

```
import classes from './file.module.css';
```

This `import` syntax is different from the `import` syntax shown at the beginning of this section for `index.css`:

```
import './index.css';
```

When importing CSS files as shown in the second snippet, the CSS code is simply merged into the `index.html` file and applied globally. When using CSS Modules instead (first code snippet), the CSS class names defined in the imported CSS file are transformed such that they are unique for the JS file that imports the CSS file.

Since the CSS class names are transformed and are therefore no longer equal to the class names you defined in the CSS file, you import an object (`classes`, in the preceding example) from the CSS file. This object exposes all transformed CSS class names under keys that match the CSS class names defined by you in the CSS file. The values of those properties are the transformed class names (as strings).

Here's a complete example, starting with a component-specific CSS file (`TextBox.module.css`):

```
.alert {
  padding: 1rem;
  border-radius: 6px;
  background-color: #f9bcb5;
  color: #480c0c;
}

.info {
  padding: 1rem;
  border-radius: 6px;
  background-color: #d6aafa;
  color: #410474;
}
```

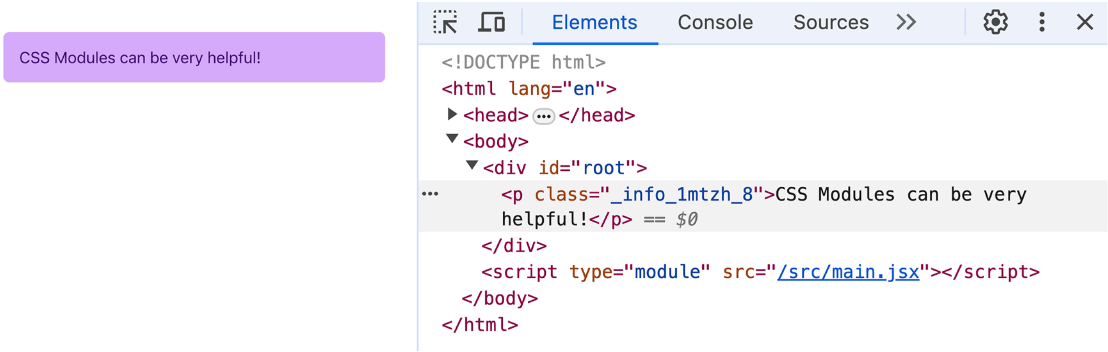
The JavaScript file (`TextBox.jsx`) for the component to which the CSS code should belong looks like this:

```
import classes from './TextBox.module.css';

function TextBox({ children, mode }) {
  let cssClasses;

  if (mode === 'alert') {
    cssClasses = classes.alert;
  } else if (mode === 'info') {
    cssClasses = classes.info;
  }

  return <p className={cssClasses}>{children}</p>;
}

export default TextBox;
```

Note

The full example code can also be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/06-styling/examples/01-css-modules-intro`.

If you inspect the rendered text element in the browser developer tools, you will note that the CSS class name applied to the `<p>` element does not match the class name specified in the `TextBox.module.css` file:



CSS Modules can be very helpful!

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
   ▼<div id="root">
…     <p class="_info_1mtzh_8">CSS Modules can be very
      helpful!</p> == $0
     </div>
     <script type="module" src="/src/main.jsx"></script>
   </body>
</html>
```

Figure 6.6: CSS class name transforms because of CSS Modules usage

This is the case because, as described previously, the class name was transformed during the build process to be unique. If any other CSS file, imported by another JavaScript file, were to define a class with the same name (`info` in this case), the styles would not clash and not overwrite each other, as the interfering class names would be transformed into different class names before being applied to the DOM elements.

Indeed, in the example provided on GitHub, you can find another `info` CSS class defined in the `index.css` file:

```
.info {
  border: 5px solid red;
}
```

That file is still imported into `main.jsx`, and hence its styles are applied globally to the entire document. Nonetheless, the `.info` styles clearly aren't affecting `<p>` rendered by `TextBox` (there is no red border around the text box in *Figure 6.6*). They aren't affecting that element because it doesn't have an `info` class anymore; the class was renamed `_info_1mtzh_8` by the build workflow (although the name you see will differ, as it contains a random element).

It's also worth noting that the `index.css` file is still imported into `main.jsx`, as shown at the beginning of this chapter. The `import` statement is not changed to `import classes from './index.css';`, nor is the CSS file called `index.module.css`.

Note, too, that you can use CSS Modules to scope styles to components and can also mix the usage of CSS Modules with regular CSS files, which are imported into JavaScript files without using CSS Modules (i.e., without scoping).

One other important aspect of using CSS Modules is that you can only use CSS class selectors (that is, in your `.module.css` files) because CSS Modules rely on CSS classes. You can write selectors that combine classes with other selectors, such as `input.invalid`, but you can't add selectors that don't use classes at all in your `.module.css` files. For example, `input { ... }` or `#some-id { ... }` selectors wouldn't work here.

CSS Modules are a very popular way of scoping styles to (React) components, and they will be used throughout many examples for the rest of this book.

The styled-components Library

The `styled-components` library is a so-called **CSS-in-JS** solution. CSS-in-JS solutions aim to remove the separation between CSS code and JavaScript code by merging them into the same file. Component styles would be defined right next to the component logic. It comes down to personal preference whether you favor separation (as enforced by using CSS files) or keeping the two languages close to each other.

Since `styled-components` is a third-party library that's not pre-installed in newly created React projects, you have to install this library as a first step if you want to use it. This can be done via `npm` (which was automatically installed together with Node.js in *Chapter 1*, *React – What and Why*):

```
npm install styled-components
```

The `styled-components` library essentially provides wrapper components around all built-in core components (as in, around `p`, `a`, `button`, `input`, and so on). It exposes all these wrapper components as **tagged templates**—JavaScript functions that aren't called like regular functions but, instead, are executed by adding backticks (a template literal) right after the function name, for example, `` doSomething`text data` ``.

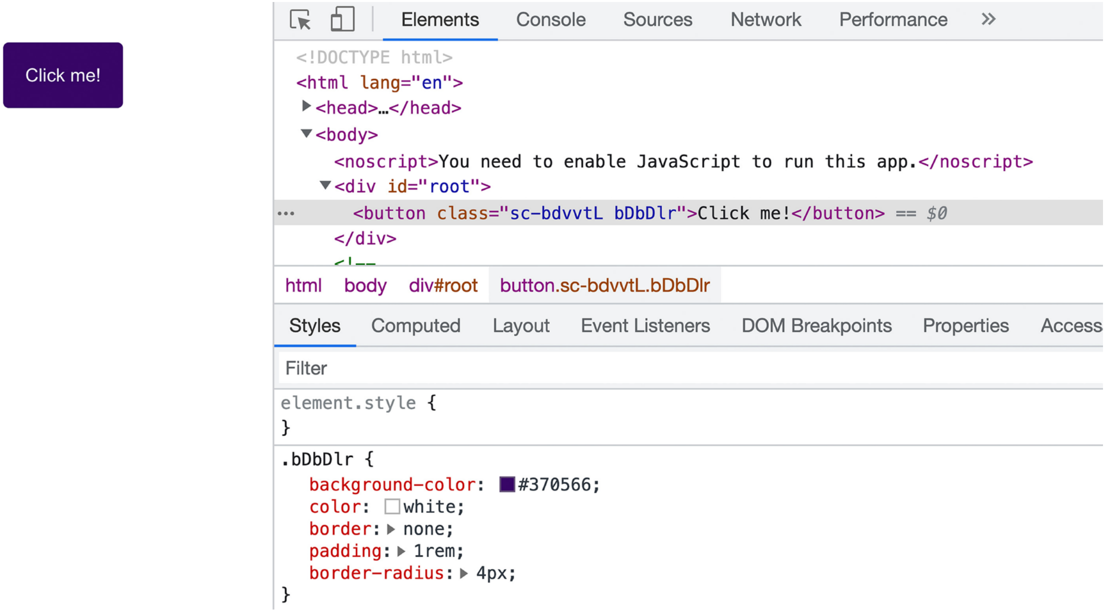
Note

Tagged templates can be confusing when you see them for the first time, especially since it's a JavaScript feature that isn't used too frequently. Chances are high that you haven't worked with them too often. It's even more likely that you have never built a custom-tagged template before. You can learn more about tagged templates in this excellent documentation on MDN at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals#tagged_templates`.

Here is a component that imports and uses `styled-components` to set and scope styling:

```
import styled from 'styled-components';

const Button = styled.button`
  background-color: #370566;
  color: white;
  border: none;
  padding: 1rem;
  border-radius: 4px;
`;

export default Button;
```

This component isn't a component function but, rather, a constant that stores the value returned by executing the `styled.button` tagged template. That tagged template returns a component function that yields a `<button>` element. The styles passed via the tagged template (i.e., inside the template literal) are applied to that returned button element. You can see this if you inspect the button in the browser's developer tools:


Click me!
Elements
Console
Sources
Network
Performance
»
html
body
div#root
button.sc-bdvvtL.bDbDlr
Styles
Computed
Layout
Event Listeners
DOM Breakpoints
Properties
Access
Filter

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
   ▼<div id="root">
...     <button class="sc-bdvvtL bDbDlr">Click me!</button> == $0
    </div>
    <!--
```

```
element.style {
}
.bDbDlr {
   background-color: ■#370566;
   color: □white;
   border: ▶ none;
   padding: ▶ 1rem;
   border-radius: ▶ 4px;
}
```

Figure 6.7: The rendered button element receives the defined component styles

In *Figure 6.7*, you can also see how the `styled-components` library applies your styles to the element. It extracts your style definitions from the tagged template string and injects them into a `<style>` element in the `<head>` section of the document. The injected styles are then applied via a class selector that is generated (and named) by the `styled-components` library. Finally, the automatically generated CSS class name is added to the element (`<button>`, in this case) by the library.

The components exposed by the `styled-components` library spread any extra props you pass to a component onto the wrapped core component. In addition, any content inserted between the opening and closing tags is also inserted between the tags of the wrapped component.

That's why the `Button` created previously can be used like this without adding any extra logic to it:

```
import Button from './components/button.jsx';

function App() {
  function handleClick() {
    console.log('This button was clicked!');
  }
  return <Button onClick={handleClick}>Click me!</Button>;
}

export default App;
```

Note

The complete example code can be found on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/06-styling/examples/02-styled-components-intro`.

You can do more with the `styled-components` library. For example, you can set styles dynamically and conditionally. This book is not primarily about that library though. It's just one of many alternatives to CSS Modules. Therefore, it is recommended that you explore the official `styled-components` documentation if you want to learn more, which you can find at `https://styled-components.com/`.

Use the Tailwind CSS Library for Styling

Scoping styles with the help of CSS modules or the `styled-component` library is a very useful and popular technique.

But no matter which approach you use, you must write all the CSS code on your own. Hence you, of course, need to know CSS.

But what if you don't? Or if you simply don't like writing CSS code?

In that case, you can use one of the many CSS libraries and frameworks available—for example, the **Bootstrap** CSS framework or the **Tailwind CSS** library. Tailwind has become a very popular styling solution for React projects (for developers who don't want to write custom CSS code).

Keep in mind that Tailwind is a CSS library that's actually not focused on React. Instead, you can use Tailwind in any web project to style your HTML code—no matter which JavaScript library or framework (if any) is being used there.

But Tailwind is a common choice for React apps, since its core philosophy plays nicely with the component-focused model of React. This is because when using Tailwind for styling, you typically compose overall styles by applying many small CSS classes to individual JSX elements:

```
function App() {
  return (
    <main
      className="bg-gray-200 text-gray-900 h-screen p-12 text-center">
      <h1 className="font-bold text-4xl">Tailwind CSS is amazing!</h1>
      <p className="text-gray-600">
        It may take a while to get used to it. But it's great for people who
don't want to write custom CSS code.
      </p>
    </main>
  );
}

export default App;
```

When first encountering code that uses Tailwind CSS, the long list of CSS classes may look intimidating and chaotic. But when working with Tailwind, you typically quickly get used to it.

Also, because Tailwind's approach offers many advantages:

- You don't need to learn CSS in detail—understanding the Tailwind syntax, which is less complex than writing CSS from scratch, suffices.
- You compose styles by combining CSS classes—similar to how you compose user interfaces from components in React.
- You don't have to switch between JSX and CSS files.
- Styling changes can be applied and tested very quickly.

As you can see in the above code snippet, the core idea of Tailwind essentially is that it provides many combinable CSS classes that each do very little. For example, the `bg-gray-200` class just sets the background color to a certain shade of gray, and nothing else.

Therefore, it's the combination of all those CSS classes that achieves a certain look, and Tailwind CSS offers many such classes that you may use and combine. You find a full list in the official documentation at `https://tailwindcss.com/docs/utility-first`.

When working with Tailwind in React projects, you can therefore build React components not just to reuse logic or JSX markup but also styles:

```
function Item({ children }) {
  return <li className='p-1 my-2 bg-stone-100'>{children}</li>;
}
```

```
function App() {
  return (
    <main className="bg-gray-200 text-gray-900 h-screen p-12 text-center">
      <h1 className="font-bold text-4xl">Tailwind CSS is amazing!</h1>
      <p className="text-gray-600">
        It may take a while to get used to it. But it's great for people who
        don't want to write custom CSS code.
      </p>
      <section className="mt-10 border border-gray-600 max-w-3xl mx-auto p-4
rounded-md bg-gray-300">
        <h2 className="font-bold text-xl">Tailwind CSS Advantages</h2>
        <ul className="mt-4">
          <Item>No CSS knowledge required</Item>
          <Item>Compose styles by combining "small" CSS classes</Item>
          <Item>
            Never leave your JSX code - no need to fiddle around in extra CSS
files
          </Item>
          <Item>Quickly test and apply changes</Item>
        </ul>
      </section>
    </main>
  );
}

export default App;
```

In this example, the `Item` component is built to reuse the Tailwind styles applied to the `<li>` element.

Note

You also find this example project on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/06-styling/examples/03-tailwind`.

If you plan on using Tailwind in your React project, you must install it as a first step. Detailed installation instructions for a broad variety of project setups can be found in the official documentation—this includes instructions for Vite projects: `https://tailwindcss.com/docs/guides/vite`.

The installation process is not as simple as just importing a CSS file but nonetheless relatively straightforward. It does require a couple of setup steps, since Tailwind needs to plug into the project build process to analyze your JSX files and produce CSS code that contains all used class names and style rules.

Besides offering many utility styles that can be combined, Tailwind also provides plenty of customization opportunities and configuration options. Therefore, entire books could be written about Tailwind alone. However, that's, of course, not what this book is about. Therefore, if you're interested in using Tailwind in your React projects, Tailwind's official documentation (see the links above) is a great place to learn more.

Using Other CSS or JavaScript Styling Libraries and Frameworks

Obviously, it comes down to personal preferences whether you want to write custom CSS code (potentially scoped with CSS Modules or `styled-components`) or whether you want to work with third-party CSS libraries, like Tailwind CSS. There is no wrong or right choice, and you'll see all kinds of approaches being used in different React projects.

The options presented in this chapter are also not exhaustive—there are other kinds of CSS and JavaScript libraries, too:

- Utility libraries that solve very specific CSS problems—independent of the fact that you're using them in a React project (for example, `Animate.css`, which helps to add animations)
- Other CSS frameworks or libraries that provide a broad variety of pre-built CSS classes that can be applied to elements to quickly achieve a certain look (for example, **Bootstrap**)
- JavaScript libraries that help with styling or specific styling aspects like animations (for example, **Framer Motion**)

Some libraries and frameworks have React-specific extensions or specifically support React, but that does not mean that you can't use libraries that don't have this.

Summary and Key Takeaways

- Standard CSS can be used to style React components and JSX elements.
- CSS files are typically directly imported into JavaScript files, which is possible thanks to the project build process, which extracts the CSS code and injects it into the document (the HTML file).
- As an alternative to global CSS styles (with `element`, `id`, `class`, or other selectors), inline styles can be used to apply styling to JSX elements.
- When using CSS classes for styling, you must use the `className` prop (not `class`).
- Styles can be set statically and dynamically or conditionally with the same syntax that is used to inject other dynamic or conditional values into JSX code—a pair of curly braces.
- Highly configurable custom components can be built by setting styles (or CSS classes) based on prop values, or by merging received prop values with other styles or class name strings.
- When using just CSS, clashing CSS class names can be a problem.
- CSS Modules solve this problem by transforming class names into unique names (per component) as part of the build workflow.
- Alternatively, third-party libraries such as `styled-components` can be used. This library is a CSS-in-JS library that also has the advantage or disadvantage (depending on your preference) of removing the separation between JS and CSS code.

- Tailwind CSS is another popular styling choice for React projects—it's a library that allows you to compose styles by combining many small CSS classes.
- Other CSS libraries or frameworks can be used as well; React does not impose any restrictions regarding that.

What's Next?

With styling covered, you're now able to build not just functional but also visually appealing user interfaces. Even if you often work with dedicated web designers or CSS experts, you still typically need to be able to set and assign styles (dynamically) that are delivered to you.

With styling being a general concept that is relatively independent of React, the next chapter will dive back into more React-specific features and topics. You will learn about **portals** and **refs**, which are two key concepts that are built into React. You will discover which problems are solved by these concepts and how the two features are used.

Test Your Knowledge!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to examples that can be found here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/06-styling/exercises/questions-answers.md`.

1. With which language are styles for React components defined?
2. Which important difference, compared to projects without React, has to be kept in mind when it comes to assigning classes to elements?
3. How can styles be assigned dynamically and conditionally?
4. What does "Scoping" mean in the context of styling?
5. How could styles be scoped to components? Briefly explain at least one concept that helps with scoping.

Apply What You Learned

You are now not only able to build interactive user interfaces but also style those user interface elements in engaging ways. You can set and change those styles dynamically or based on conditions.

In this section, you will find two activities that allow you to apply your newly gained knowledge in combination with what you learned in previous chapters.

Activity 6.1: Providing Input Validity Feedback upon Form Submission

In this activity, you will build a basic form that allows users to enter an email address and a password. The provided input of each input field is validated, and the validation result is stored (for each individual input field).

The aim of this activity is to add some general form styling and some conditional styling that becomes active once an invalid form has been submitted. The exact styles are up to you, but to highlight invalid input fields, the background color of the affected input field must be changed, as well as its border color and the text color of the related label.

The steps are as follows:

1. Create a new React project and add a form component to it.
2. Output the form component in the project's root component.
3. In the form component, output a form that contains two input fields: one for entering an email address and one for entering a password.
4. Add labels to the input fields.
5. Store the entered values and check their validity upon form submission (you can be creative in forming your own validation logic).
6. Pick appropriate CSS classes from the provided `index.css` file (alternatively, you can write your own classes as well).
7. Add them to the invalid input fields and their labels once invalid values have been submitted.

The final user interface should look like this:

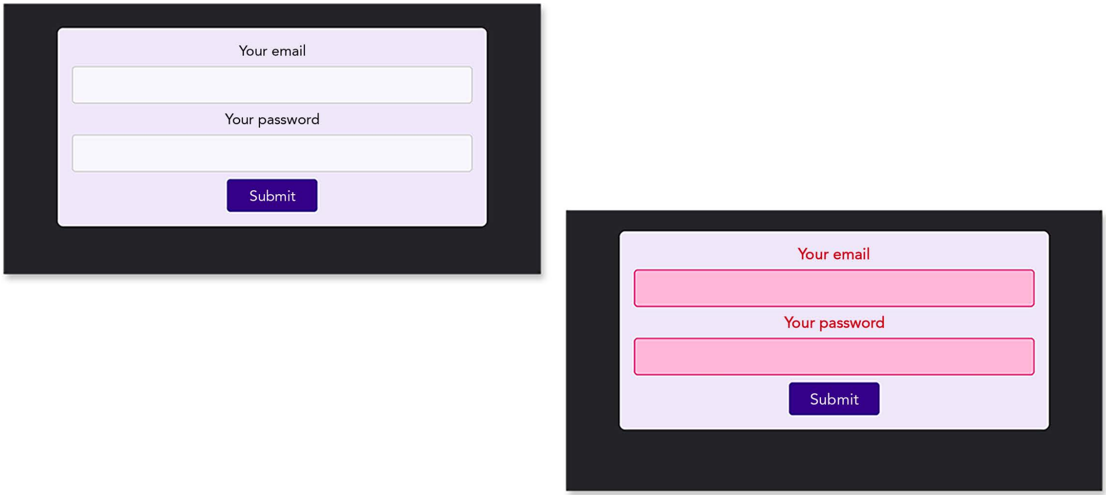


Figure 6.8: The final user interface with invalid input values highlighted in red

Since this book is not about CSS and you may not be a CSS expert, you can use the `index.css` file from the solution and focus on the React logic to apply appropriate CSS classes to JSX elements.

Note

All code files used for this activity, and a full solution, can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/06-styling/activities/practice-1`.

Activity 6.2: Using CSS Modules for Style Scoping

In this activity, you'll take the final app built in *Activity 6.1* and adjust it to use CSS Modules. The goal is to migrate all component-specific styles into a component-specific CSS file, which uses CSS Modules for style scoping.

The final user interface therefore looks the same as it did in the previous activity. However, the styles will be scoped to the `Form` component so that clashing class names won't interfere with styling.

The steps are as follows:

1. Finish the previous activity or take the finished code from GitHub.
2. Identify the styles belonging specifically to the `Form` component and move them into a new, component-specific CSS file.
3. Change CSS selectors to class name selectors and add classes to JSX elements as needed (this is because CSS Modules require class name selectors).
4. Use the component-specific CSS file as explained throughout this chapter and assign the CSS classes to the appropriate JSX elements.

Note

All code files used for this activity, and a full solution, can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/06-styling/activities/practice-2`.

7

Portals and Refs

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Use direct DOM element access to interact with elements
- Expose the functions and data of your components to other components
- Control the position of rendered JSX elements in the DOM

Introduction

React.js is all about building user interfaces, and, in the context of this book, it's specifically about building web user interfaces.

Web user interfaces are ultimately all about the **Document Object Model** (**DOM**). You can use JavaScript to read or manipulate the DOM. This is what allows you to build interactive websites: you can add, remove, or edit DOM elements after a page has been loaded. This can be used to add or remove overlay windows or to read values entered into input fields.

This was discussed in *Chapter 1*, *React – What and Why*, and, as you learned there, React is used to simplify this process. Instead of manipulating the DOM or reading values from DOM elements manually, you can use React to describe the desired state. React then takes care of the steps needed to achieve this desired state.

However, there are scenarios and use cases wherein, despite using React, you still want to be able to directly reach out to specific DOM elements—for example, to read a value entered by a user into an input field, or if you're not happy with the position of a newly inserted element in the DOM that was chosen by React.

React provides certain functionalities that help you in exactly these kinds of situations: **Portals** and **Refs.** Even though directly manipulating the DOM will still not be a great idea, these tools, as you will learn throughout this chapter, can help with reading DOM element values or with changing the DOM structure without working against React.

A World without Refs

Consider the following example: you have a website that renders an input field, requesting a user's email address. It could look something like this:

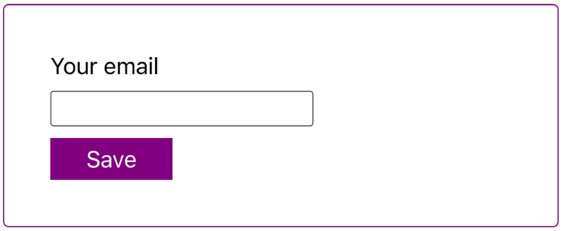


Figure 7.1: An example form with an email input field

The code for the component that's responsible for rendering the form and handling the entered email address value might look like this:

```
function EmailForm() {
  const [enteredEmail, setEnteredEmail] = useState('');

  console.log(enteredEmail);

  function handleUpdateEmail(event) {
    setEnteredEmail(event.target.value);
  }

  function handleSubmitForm(event) {
    event.preventDefault();
    // could send enteredEmail to a backend server
  }

  return (
    <form className={classes.form} onSubmit={handleSubmitForm}>
      <label htmlFor="email">Your email</label>
      <input type="email" id="email" onChange={handleUpdateEmail} />
      <button>Save</button>
    </form>
  );
}
```

As you can see, this example uses the `useState()` Hook, combined with the `change` event, to register keystrokes in the `email` input field and store the entered value.

This code works fine, and there is nothing wrong with having this kind of code in your application. But adding the extra event listener and state, as well as adding the function to update the state whenever the `change` event is triggered, is quite a bit of boilerplate code for one simple task: getting the entered email address.

The preceding code snippet does nothing else with the email address other than submit it. In other words, the only reason for using the `enteredEmail` state in the example is to read the entered value.

Even though the `enteredEmail` is only required in the `handleSubmitForm()` function, React will re-execute the `EmailForm` component function for every `enteredEmail` state updated, i.e., for every keystroke in the `<input>` field. This is also not ideal since it leads to lots of unnecessary code execution and hence potential performance issues.

In scenarios such as this, quite a bit of code (and maybe performance) could be saved if you fell back to some vanilla JavaScript logic:

```
const emailInputEl = document.getElementById('email');
const enteredEmailVal = emailInputEl.value;
```

These two lines of code (which could be merged into one line theoretically) allow you to get hold of a DOM element and read the currently stored value.

The problem with this kind of code is that it does not use React. And if you're building a React app, you should really stick to React when working with the DOM. Don't start blending your own vanilla JavaScript code *that accesses the DOM* into the React code.

This can lead to unintended behaviors or bugs, especially if you start manipulating the DOM. It could lead to bugs because React would not be aware of your changes in that case; the actual rendered UI would not be in sync with React's assumed UI. Even if you're just reading from the DOM, it's a good habit to not even start merging vanilla JavaScript DOM access methods with your React code.

To still allow you to get hold of DOM elements and read values, as shown above, React gives you a special concept that you can use: **Refs**.

Ref stands for reference, and it's a feature that allows you to store references to values—for example, to DOM elements from inside a React component. The preceding vanilla JavaScript code would do the same (it also gives you access to a rendered element), but when using Refs, you can get access without mixing vanilla JavaScript code into your React code.

Refs can be created using a special React Hook called the `useRef()` Hook.

This Hook can be executed to generate a `ref` object:

```
import { useRef } from 'react';

function EmailForm() {
  const emailRef = useRef(null);
  // other code ...
};
```

This generated Ref object, `emailRef` in the preceding example, is initially set to null but can then be assigned to any JSX element. This assignment is done via a special prop (the `ref` prop) that is automatically supported by every JSX element:

```
return (
  <form className={classes.form} onSubmit={handleSubmitForm}>
    <label htmlFor="email">Your email</label>
    <input
      ref={emailRef}
      type="email"
      id="email"
    />
    <button>Save</button>
  </form>
);
```

Just like the `key` prop introduced in *Chapter 5*, *Rendering Lists and Conditional Content*, the `ref` prop is provided by React. The `ref` prop wants a Ref object, i.e., one that was created via `useRef()`.

In this example, `useRef()` receives `null` as an initial value since it's technically not yet assigned to the DOM element when the component function executes for the first time. It's only after that initial component render cycle that the connection will be established. Therefore, after this first component function execution, the value stored in the Ref will be the underlying DOM object of the `<input>` element in this example.

With that Ref object created and assigned, you can then use it to get access to the connected JSX element (to the `<input>` element in this example). There's just one important thing to note: to get hold of the connected element, you must access a special `current` prop on the created Ref object. This is required because React stores the value assigned to the Ref object in a nested object, accessible via the `current` property, as shown here:

```
function handleSubmitForm(event) {
  event.preventDefault();
  const enteredEmail = emailRef.current.value; // .current is mandatory!
  // could send enteredEmail to a backend server
};
```

`emailRef.current` yields the underlying DOM object that was rendered for the connected JSX element. In this case, it therefore allows access to the input element DOM object. Since that DOM object has a `value` property, this `value` property can be accessed without issue.

Note

For further information on this topic, see `https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#attributes`.

With this kind of code, you can read the value from the DOM element without having to use `useState()` and an event listener. The final component code therefore becomes quite a bit leaner:

```
function EmailForm() {
  const emailRef = useRef(null);

  function handleSubmitForm(event) {
    event.preventDefault();
    const enteredEmail = emailRef.current.value;
    // could send enteredEmail to a backend server
  }

  return (
    <form className={classes.form} onSubmit={handleSubmitForm}>
      <label htmlFor="email">Your email</label>
      <input
        ref={emailRef}
        type="email"
        id="email"
      />
      <button>Save</button>
    </form>
  );
}
```

Refs versus State

Since Refs can be used to get quick and easy access to DOM elements, the question that might come up is whether you should always use Refs instead of state.

The clear answer to this question is “no.”

Refs can be a very good alternative in use cases like the one shown above, when you need read access to an element. This is very often the case when dealing with user input. In general, Refs can replace state if you’re just accessing some value to read it when some function (a form submit handler function, for example) is executed. As soon as you need to change values and those changes must be reflected in the UI (for example, by rendering some conditional content), Refs are out of the game.

In the example above, if, besides getting the entered value, you'd also like to reset (i.e., clear) the email input after the form was submitted, you should use state again. While you could reset the input with the help of a Ref, you should not do that. You would start manipulating the DOM, and only React should do that—with its own, internal methods, based on the declarative code you provide to React.

You should avoid resetting the email input like this:

```
function EmailForm() {
  const emailRef = useRef(null);

  function handleSubmitForm(event) {
    event.preventDefault();
    const enteredEmail = emailRef.current.value;
    // could send enteredEmail to a backend server

    emailRef.current.value = ''; // resetting the input value
  }

  return (
    <form className={classes.form} onSubmit={handleSubmitForm}>
      <label htmlFor="email">Your email</label>
      <input
        ref={emailRef}
        type="email"
        id="email"
      />
      <button>Save</button>
    </form>
  );
}
```

Instead, you should reset it by using React's state concept and by following the declarative approach embraced by React:

```
function EmailForm() {
  const [enteredEmail, setEnteredEmail] = useState('');

  function handleUpdateEmail(event) {
    setEnteredEmail(event.target.value);
  }

  function handleSubmitForm(event) {
    event.preventDefault();
    // could send enteredEmail to a backend server

    // reset by setting the state + using the value prop below
    setEnteredEmail('');
  }

  return (
    <form className={classes.form} onSubmit={handleSubmitForm}>
      <label htmlFor="email">Your email</label>
      <input
        type="email"
        id="email"
        onChange={handleUpdateEmail}
        value={enteredEmail}
      />
      <button>Save</button>
    </form>
  );
}
```

> **Note**
>
> As a rule, you should simply try to avoid writing imperative code in React projects. Instead, tell React how the final UI should look and let React figure out how to get there.
>
> Reading values via Refs is an acceptable exception, and manipulating DOM elements (with or without Refs, e.g., by directly selecting DOM nodes via `document.getElementById()` or similar) should be avoided. A rare exception is a scenario such as calling `focus()` on an input element DOM object because methods like `focus()` don't typically cause any DOM changes that could break the React app.

Using Refs for More than DOM Access

Accessing DOM elements (for reading values) is one of the most common use cases for using Refs. As shown above, it can help you reduce code in certain situations.

But Refs are more than just "element connection bridges;" they are objects that can be used to store all kinds of values—not just pointers at DOM objects. You can, for example, also store strings or numbers or any other kind of value in a Ref:

```
const passwordRetries = useRef(0);
```

You can pass an initial value to `useRef()` (`0` in this example) and then access or change that value at any point in time inside the component to which the Ref belongs:

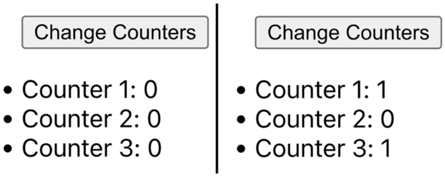
```
passwordRetries.current = 1;
```

However, you still have to use the `current` property to read and change the stored value, because, as mentioned above, this is where React will store the actual value that belongs to the Ref.

This can be useful for storing data that should "survive" component re-evaluations. As you learned in *Chapter 4*, *Working with Events and State*, React will execute component functions every time the state of a component changes. Since the function is executed again, any data stored in function-scoped variables would be lost. Consider the following example:

```
function Counters() {
  const [counter1, setCounter1] = useState(0);
  const counterRef = useRef(0);
  let counter2 = 0;

  function handleChangeCounters() {
    setCounter1(1);
    counter2 = 1;
    counterRef.current = 1;
  };

  return (
    <>
      <button onClick={handleChangeCounters}>Change Counters</button>
      <ul>
        <li>Counter 1: {counter1}</li>
        <li>Counter 2: {counter2}</li>
        <li>Counter 3: {counterRef.current}</li>
      </ul>
    </>
  );
};
```

In this example, counters 1 and 3 would change to 1 once the button is clicked. However, counter 2 would remain zero, even though the `counter2` variable gets changed to a value of 1 in `handleChangeCounters` as well:



- Counter 1: 0
- Counter 2: 0
- Counter 3: 0

- Counter 1: 1
- Counter 2: 0
- Counter 3: 1

Figure 7.2: Only two of the three counter values changed

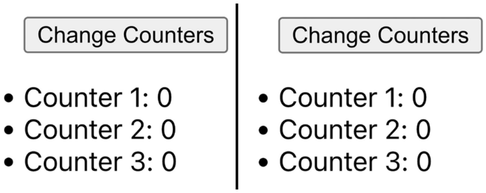
In this example, it should be expected that the state value changes and the new value is reflected in the updated user interface. That is the whole idea behind state, after all.

The Ref (`counterRef`) also keeps its updated value across component re-evaluations, though. That's the behavior described above: Refs are not reset or cleared when the surrounding component function is executed again. The vanilla JavaScript variable (`counter2`) does not keep its value. Even though it is changed in `handleChangeCounters`, a new variable is initialized when the component function is executed again; thus the updated value (1) is lost.

In this example, it might again look like Refs can replace state, but the example actually shows very well why that is **not** the case. Try replacing `counter1` with another Ref (so that there is no state value left in the component) and clicking the button:

```
import { useRef } from 'react';

function Counters() {
  const counterRef1 = useRef(0);
  const counterRef2 = useRef(0);
  let counter2 = 0;

  function handleChangeCounters() {
    counterRef1.current = 1;
    counter2 = 1;
    counterRef2.current = 1;
  }

  return (
    <>
      <button onClick={handleChangeCounters}>Change Counters</button>
      <ul>
        <li>Counter 1: {counterRef1.current}</li>
        <li>Counter 2: {counter2}</li>
        <li>Counter 3: {counterRef2.current}</li>
      </ul>
    </>
  }
);


export default Counters;
```

Nothing will change on the page because, while the button click is registered and the `handleChangeCounters` function is executed, no state change is initiated, and state changes (initiated via the `setXYZ` state updating function calls) are the triggers that cause React to re-evaluate a component. Changes to Ref values do **not** do that.



- Counter 1: 0
- Counter 2: 0
- Counter 3: 0

- Counter 1: 0
- Counter 2: 0
- Counter 3: 0

Figure 7.3: The counter values don't change

As you can tell, changing Ref values does not trigger component functions to be executed again—state, on the other hand, does. However, if a component function runs again (due to a state change), Ref values are kept around and not dropped.

Therefore, if you have data that should survive component re-evaluations but should not be managed as state (because changes to that data should not cause the component to be re-evaluated when changed), you could use a Ref:

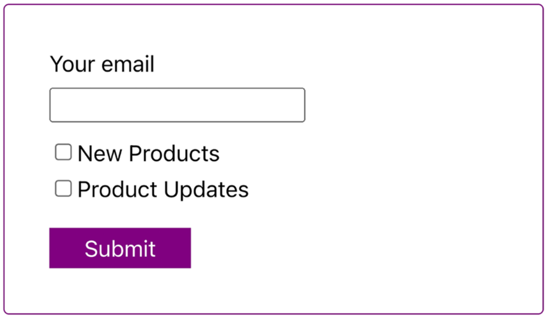
```
const passwordRetries = useRef(0);
// later in the component ...
passwordRetries.current = 1; // changed from 0 to 1
// later ...
console.log(passwordRetries.current); // prints 1, even if the component changed
```

This is not a feature that's used frequently, but it can be helpful from time to time. In all other cases, use normal state values.

Refs in Custom Components

Refs cannot just be used to access DOM elements. You can also use them to access React components—including your own components.

This can sometimes be useful. Consider this example: you have a `<Form>` component that contains a nested `<Preferences>` component. The latter component is responsible for displaying two checkboxes, asking the user for their newsletter preferences:



☐ New Products
☐ Product Updates

Figure 7.4: A newsletter sign-up form that shows two checkboxes to set newsletter preferences

The code of the `Preferences` component could look like this:

```
function Preferences() {
  const [wantsNewProdInfo, setWantsNewProdInfo] = useState(false);
  const [wantsProdUpdateInfo, setWantsProdUpdateInfo] = useState(false);

  function handleChangeNewProdPref() {
    setWantsNewProdInfo((prevPref) => !prevPref);
  }

  function handleChangeUpdateProdPref() {
    setWantsProdUpdateInfo((prevPref) => !prevPref);
  }

  return (
    <div className={classes.preferences}>
      <label>
        <input
          type="checkbox"
          id="pref-new"
          checked={wantsNewProdInfo}
          onChange={handleChangeNewProdPref}
        />
        <span>New Products</span>
      </label>
      <label>
        <input
          type="checkbox"
          id="pref-updates"
          checked={wantsProdUpdateInfo}
          onChange={handleChangeUpdateProdPref}
        />
        <span>Product Updates</span>
      </label>
    </div>
  );
};
```

As you can see, it's a basic component that essentially outputs the two checkboxes, adds some styling, and keeps track of the selected checkbox via state.

The `Form` component code could look like this:

```
function Form() {
  function handleSubmit(event) {
    event.preventDefault();
  }

  return (
    <form className={classes.form} onSubmit={handleSubmit}>
      <div className={classes.formControl}>
        <label htmlFor="email">Your email</label>
        <input type="email" id="email" />
      </div>
      <Preferences />
      <button>Submit</button>
    </form>
  );
}
```

Now imagine that upon form submission (inside of the `handleSubmit` function), the `Preferences` should be reset (i.e., no checkbox is selected anymore). In addition, prior to resetting, the selected values should be read and used in the `handleSubmit` function.

This would be straightforward if the checkboxes were not put into a separate component. If the entire code and JSX markup reside in the `Form` component, state could be used in that component to read and change the values. But this is not the case in this example, and rewriting the code just because of this problem sounds like an unnecessary restriction.

Fortunately, Refs can help in this situation.

You can expose features (for example, functions or state values) of a component to other components via Refs. Refs can essentially be used as a *communication device* between two components, just as they were used as a *communication device* with a DOM element in the previous sections.

Conveniently, your custom components can receive a ref as a regular prop:

```
function Preferences(props) { // or function Preferences({ ref }) {}
  // can use props.ref in here
  // component code ...
};

export default Preferences;
```

You could therefore use this `Preferences` component and pass a `ref` to it:

```
function Form() {
  const preferencesRef = useRef(null);

  return <Preferences ref={preferencesRef} />;
}
```

It's important to note that this code only works when using React 19 or higher. When working with an older React version, passing Refs as regular props to components is unfortunately not supported. In such projects, you would have to wrap the component function that should receive a Ref with a special `forwardRef()` function that's provided by React.

Therefore, in React projects using React 18 or older, to receive and use Refs, you must wrap the receiving component (`Preferences`, in this example) with `forwardRef()`.

This can be done like this:

```
const Preferences = forwardRef((props, ref) => {
  // component code ...
});

export default Preferences;
```

This looks slightly different than all the other components in this book because an arrow function is used instead of the `function` keyword. You can always use arrow functions instead of "normal functions", but here it's helpful to switch as it makes wrapping the function with `forwardRef()` very easy. Alternatively, you could stick to the `function` keyword and wrap the function like this:

```
function Preferences(props, ref) {
 // component code ...
};

export default forwardRef(Preferences);
```

It is up to you which syntax you prefer. Both work and both are commonly used in React projects.

The interesting part about this code is that the component function now receives **two** parameters instead of one. Besides receiving `props`, which component functions always do, it now also receives a special `ref` parameter. And this parameter is only received because the component function is wrapped with `forwardRef()`.

This `ref` parameter will contain any `ref` value set by the component using the `Preferences` component. For example, the `Form` component could set a `ref` parameter on `Preferences` like this:

```
function Form() {
  const preferencesRef = useRef({});
  function handleSubmit(event) {
    // other code ...
  }

  return (
    <form className={classes.form} onSubmit={handleSubmit}>
      <div className={classes.formControl}>
        <label htmlFor="email">Your email</label>
        <input type="email" id="email" />
      </div>
      <Preferences ref={preferencesRef} />
      <button>Submit</button>
    </form>
  );
}
```

Again, `useRef()` is used to create a `ref` object (`preferencesRef`), and that object is then passed via the special `ref` prop to the `Preferences` component. The created Ref receives a default value of an empty object (`{}`); it's this object that can then be accessed via `ref.current`. In the `Preferences` component, the `ref` value can either be received and extracted like a regular prop (React >= 19) or must be accessed with the help of React's `forwardRef()` function. In that case, it's received via this second `ref` parameter, which exists because of `forwardRef()`.

But what's the benefit of that? How can this `preferencesRef` object now be used inside `Preferences` to enable cross-component interaction?

Since `ref` is an object that is never replaced, even if the component in which it was created via `useRef()` is re-evaluated (see previous sections above), the receiving component can assign properties and methods to that object and the creating component can then use these methods and properties. The `ref` object is therefore used as a communication vehicle.

In this example, the `Preferences` component could be changed like this to use the `ref` object:

```
function Preferences(props) { // wrap with forwardRef() for React < 19
  const { ref } = props; // Extracting ref prop
  const [wantsNewProdInfo, setWantsNewProdInfo] = useState(false);
  const [wantsProdUpdateInfo, setWantsProdUpdateInfo] = useState(false);

  function handleChangeNewProdPref () {
    setWantsNewProdInfo((prevPref) => !prevPref);
  }

  function handleChangeUpdateProdPref() {
    setWantsProdUpdateInfo((prevPref) => !prevPref);
  }

  function reset() {
    setWantsNewProdInfo(false);
    setWantsProdUpdateInfo(false);
  }

  ref.current.reset = reset;
  ref.current.selectedPreferences = {
    newProductInfo: wantsNewProdInfo,
    productUpdateInfo: wantsProdUpdateInfo,
  };

  // also return JSX code (has not changed) ...
});
```

In `Preferences`, both the state values and a pointer at a newly added `reset` function are stored in the received `ref` object. `ref.current` is used since the object created by React (when using `useRef()`) always has such a `current` property, and that property should be used to store the actual values in `ref`.

Since `Preferences` and `Form` operate on the same object that's stored in the `ref` object, the properties and methods assigned to the object in `Preferences` can also be used in `Form`:

```
function Form() {
  const preferencesRef = useRef({});

  function handleSubmit(event) {
    event.preventDefault();

    console.log(preferencesRef.current.selectedPreferences); // reading a value
    preferencesRef.current.reset(); // executing a function stored in
Preferences
  }

  return (
    <form className={classes.form} onSubmit={handleSubmit}>
      <div className={classes.formControl}>
        <label htmlFor="email">Your email</label>
        <input type="email" id="email" />
      </div>
      <Preferences ref={preferencesRef} />
      <button>Submit</button>
    </form>
  );
}
```

By using Refs like this, a parent component (`Form`, in this case) is able to interact with some child component (for instance, `Preferences`) in an imperative way—meaning properties can be accessed and methods called to manipulate the child component (or, to be precise, to trigger some internal functions and behavior inside the child component).

Note

React also provides an `useImperativeHandle()` Hook that may be used to expose data or functions from custom components.

Technically, you don't need to use this Hook, as the above examples prove. You can communicate between components via Refs without any extra Hooks.

But you might want to consider using `useImperativeHandle()` since it will handle scenarios like missing `ref` values (i.e., if no `ref` value is provided). You can learn more about the usage of this (arguably niche) Hook in the official documentation: `https://react.dev/reference/react/useImperativeHandle`.

Controlled versus Uncontrolled Components

Passing Refs to custom components (via props or `forwardRef()`) is a method that can be used to allow the `Form` and `Preferences` components to work together. But even though it might look like an elegant solution at first, it should typically not be your default solution for this kind of problem.

Using Refs, as shown in the example above, leads to more imperative code in the end. It's imperative code because instead of defining the desired user interface state via JSX (which would be declarative), individual step-by-step instructions are added in JavaScript.

If you revisit *Chapter 1*, *React – What and Why* (the *The Problem with Vanilla JavaScript* section), you'll see that code such as `preferencesRef.current.reset()` (from the example above) looks quite similar to instructions such as `buttonElement.addEventListener(…)` (example from *Chapter 1*). Both examples use imperative code and should be avoided for the reasons mentioned in *Chapter 1* (writing step-by-step instructions leads to inefficient micro-management and often unnecessarily complex code).

Inside the `Form` component, the `reset()` function of `Preferences` is invoked. Hence, the code describes the desired action that should be performed (instead of the expected outcome). Typically, when working with React you should strive to describe the desired (UI) state instead. Remember, when working with React, that you should write declarative, rather than imperative, code.

When using Refs to read or manipulate data as shown in the previous sections of this chapter, you are building so-called **uncontrolled components**. The components are considered "uncontrolled" because React is not directly controlling the UI state. Instead, values are read from other components or the DOM. It's therefore the DOM that controls the state (e.g., a state such as the value entered by a user into an input field).

As a React developer, you should try to minimize the use of uncontrolled components. It's absolutely fine to use Refs to save some code if you only need to gather some entered values. But as soon as your UI logic becomes more complex (for example, if you also want to clear user input), you should go for **controlled components** instead.

Doing so is quite straightforward: a component becomes controlled as soon as React manages the state. In the case of the `EmailForm` component from the beginning of this chapter, the controlled component approach was shown before Refs were introduced. Using `useState()` to store the user's input (and update the state with every keystroke) meant that React was in full control of the entered value.

For the previous example, the `Form` and `Preferences` components, switching to a controlled component approach could look like this:

```
function Preferences({newProdInfo, prodUpdateInfo, onUpdateInfo}) {
  return (
    <div className={classes.preferences}>
      <label>
        <input
          type="checkbox"
          id="pref-new"
          checked={newProdInfo}
          onChange={onUpdateInfo.bind(null, 'pref-new')}
        />
        <span>New Products</span>
      </label>
      <label>
        <input
          type="checkbox"
          id="pref-updates"
          checked={prodUpdateInfo}
          onChange={onUpdateInfo.bind(null, 'pref-updates')}
        />
        <span>Product Updates</span>
      </label>
    </div>
  );
};
```

In this example, the `Preferences` component stops managing the checkbox state and instead receives props from its parent component (the `Form` component).

`bind()` is used on the `onUpdateInfo` prop (which will receive a function as a value) to *pre-configure* the function for future execution. `bind()` is a default JavaScript method that can be called on any JavaScript function to control which arguments will be passed to that function once it's invoked in the future.

Note

You can learn more about this JavaScript feature at `https://academind.com/tutorials/function-bind-event-execution`.

The `Form` component now manages the checkbox states, even though it doesn't directly contain the checkbox elements. But it now begins to control the `Preferences` component and its internal state, hence turning `Preferences` into a controlled component instead of an uncontrolled one:

```
function Form() {
  const [wantsNewProdInfo, setWantsNewProdInfo] = useState(false);
  const [wantsProdUpdateInfo, setWantsProdUpdateInfo] = useState(false);

  function handleUpdateProdInfo(selection) {
    // using one shared update handler function is optional
    // you could also use two separate functions (passed to Preferences) as
props
    if (selection === 'pref-new') {
      setWantsNewProdInfo((prevPref) => !prevPref);
    } else if (selection === 'pref-updates') {
      setWantsProdUpdateInfo((prevPref) => !prevPref);
    }
  }

  function reset() {
    setWantsNewProdInfo(false);
    setWantsProdUpdateInfo(false);
  }

  function handleSubmit(event) {
    event.preventDefault();
    // state values can be used here
    reset();
  }

  return (
    <form className={classes.form} onSubmit={handleSubmit}>
      <div className={classes.formControl}>
        <label htmlFor="email">Your email</label>
        <input type="email" id="email" />
      </div>
      <Preferences
```

```
        newProdInfo={wantsNewProdInfo}
        prodUpdateInfo={wantsProdUpdateInfo}
        onUpdateInfo={handleUpdateProdInfo}
      />
      <button>Submit</button>
    </form>
  );
}
```

`Form` manages the checkbox selection state, including resetting the state via the `reset()` function, and passes the managed state values (`wantsNewProdInfo` and `wantsProdUpdateInfo`) as well as the `handleUpdateProdInfo` function, which updates the state values, to `Preferences`. The `Form` component now controls the `Preferences` component.

If you go through the two code snippets above, you'll notice that the final code is once again purely declarative. Across all components, state is managed and used to declare the expected user interface.

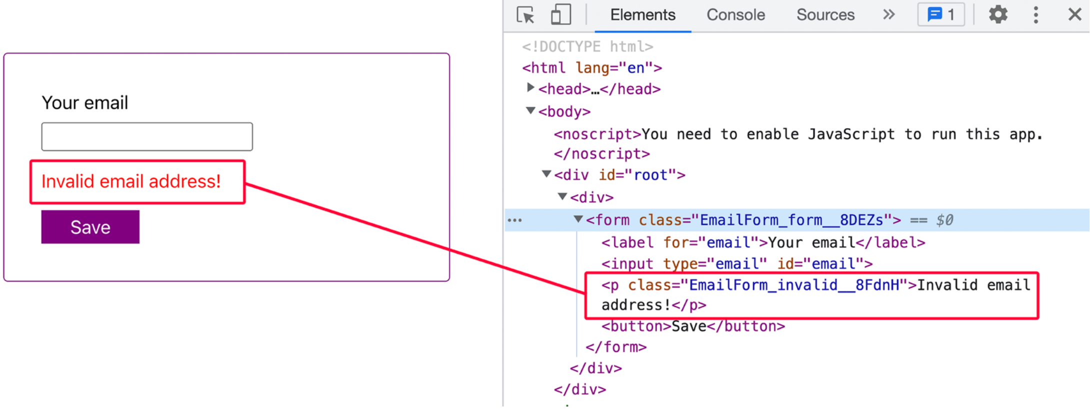
It is considered a good practice to go for controlled components in most cases. If you are only extracting some entered user input values, however, then using Refs and creating an uncontrolled component is absolutely fine.

React and Where Things End up in the DOM

Leaving the topic of Refs, there is one other important React feature that can help with influencing (indirect) DOM interaction: **Portals**.

When building user interfaces, you sometimes need to display elements and content conditionally. This was already covered in *Chapter 5*, *Rendering Lists and Conditional Content*. When rendering conditional content, React will inject that content into the place in the DOM where the overall component (in which the conditional content is defined) is located.

For example, when showing a conditional error message below an input field, that error message is right below the input in the DOM:

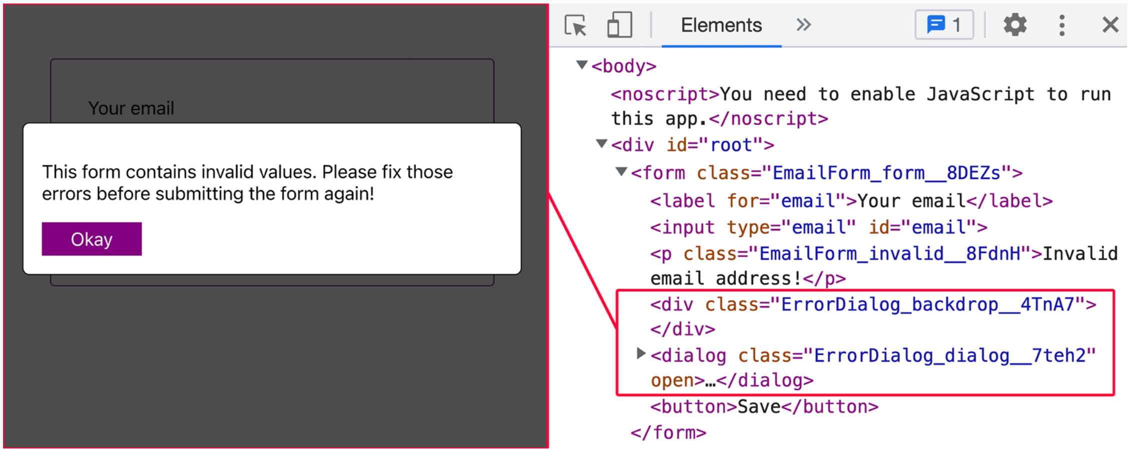


```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
    <noscript>You need to enable JavaScript to run this app.
    </noscript>
   ▼<div id="root">
     ▼<div>
       ▼<form class="EmailForm_form__8DEZs"> == $0
          <label for="email">Your email</label>
          <input type="email" id="email">
          <p class="EmailForm_invalid__8FdnH">Invalid email
          address!</p>
          <button>Save</button>
        </form>
      </div>
    </div>
```

Figure 7.5: The error message DOM element sits right below the <input> it belongs to

This behavior makes sense. Indeed, it would be pretty irritating if React were to start inserting DOM elements in random places. But in some scenarios, you may prefer a (conditional) DOM element to be inserted in a different place in the DOM—for example, when working with overlay elements such as error dialogs.

In the preceding example, you could add logic to ensure that an error dialog is presented to the user if the form is submitted with an invalid email address. This could be implemented with logic similar to the `"Invalid email address!"` error message, and therefore the dialog element would, of course, also be injected dynamically into the DOM:



This form contains invalid values. Please fix those errors before submitting the form again!

```
▼<body>
   <noscript>You need to enable JavaScript to run
   this app.</noscript>
  ▼<div id="root">
    ▼<form class="EmailForm_form__8DEZs">
       <label for="email">Your email</label>
       <input type="email" id="email">
       <p class="EmailForm_invalid__8FdnH">Invalid
       email address!</p>
       <div class="ErrorDialog_backdrop__4TnA7">
       </div>
      ▶<dialog class="ErrorDialog_dialog__7teh2"
       open>…</dialog>
       <button>Save</button>
     </form>
```

Figure 7.6: The error dialog and its backdrop are injected into the DOM

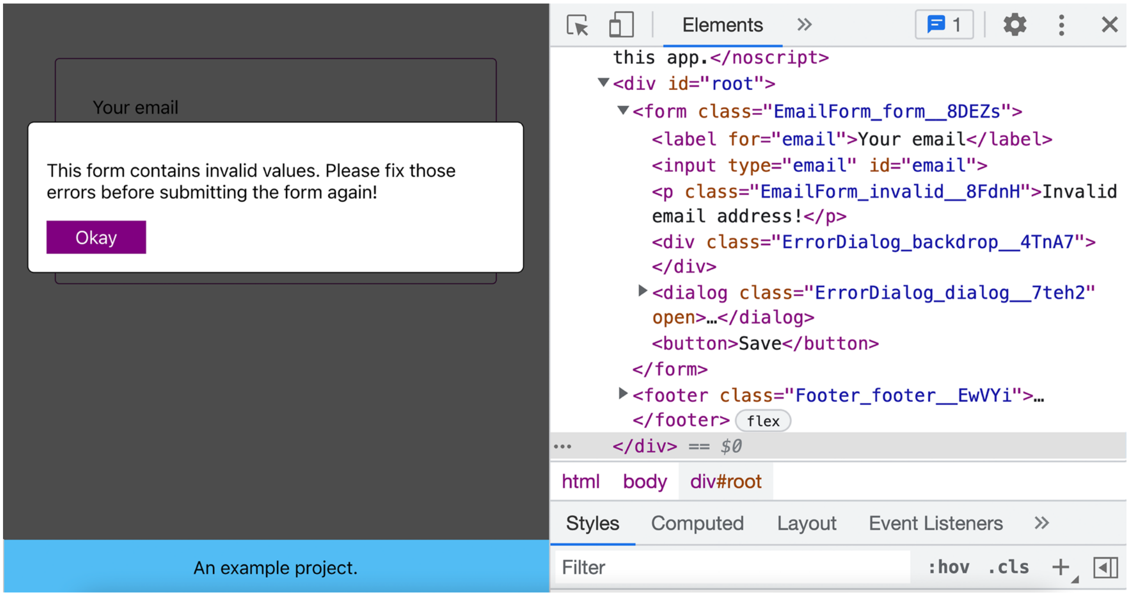
In this screenshot, the error dialog is opened as an overlay above a backdrop element, which is itself added so that it acts as an overlay to the rest of the user interface.

Note

The appearance is handled entirely by CSS, and you can take a look at the complete project (including the styling) here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/07-portals-refs/examples/05-portals-problem`.

This example works and looks fine. However, there is room for improvement.

Semantically, it doesn't entirely make sense to have the overlay elements injected somewhere nested into the DOM next to the `<input>` element. It would make more sense for overlay elements to be closer to the root of the DOM (in other words, to be direct child elements of `<div id="root">` or even `<body>`), instead of being children of `<form>`. And it's not just a semantic problem. If the example app contains other overlay elements, those elements might clash with each other, like this:



Your email

This form contains invalid values. Please fix those errors before submitting the form again!

An example project.

```
    this app.</noscript>
   ▼<div id="root">
     ▼<form class="EmailForm_form__8DEZs">
        <label for="email">Your email</label>
        <input type="email" id="email">
        <p class="EmailForm_invalid__8FdnH">Invalid
        email address!</p>
        <div class="ErrorDialog_backdrop__4TnA7">
        </div>
       ▶<dialog class="ErrorDialog_dialog__7teh2"
        open>…</dialog>
        <button>Save</button>
      </form>
     ▶<footer class="Footer_footer__EwVYi">…
      </footer> flex
•••  </div> == $0
```

Figure 7.7: The <footer> element at the bottom is visible above the backdrop

In this example, the `<footer>` element at the bottom ("An example project") is not hidden or grayed out by the backdrop that belongs to the error dialog. The reason for that is that the footer also has some CSS styling attached that turns it into a de facto overlay (because of `position: fixed` and `left` + `bottom` being used in its CSS styles).

As a solution to this problem, you could tweak some CSS styles and, for example, use the `z-index` CSS property to control overlay levels. However, it would be a cleaner solution if the overlay elements (i.e., the `<div>` backdrop and the `<dialog>` error elements) were inserted into the DOM in a different place—for example, at the very end of the `<body>` element (but as direct children to `<body>`).

And that's exactly the kind of problem React **Portals** help you solve.

Portals to the Rescue

A **Portal**, in React's world, is a feature that allows you to instruct React to insert a DOM element in a different place than where it would normally be inserted.

Considering the example shown above, this portal feature can be used to *tell* React to not insert the `<dialog>` error and the `<div>` backdrop that belongs to the dialog inside the `<form>` element, but to instead insert those elements at the end of the `<body>` element.

To use this portal feature, you first must define a place wherein elements can be inserted (an "injection hook"). This can be done in the HTML file that belongs to the React app (i.e., `index.html`). There, you can add a new element (for example, a `<div>` element) somewhere in the `<body>` element:

```
<body>
  <div id="root"></div>
  <div id="dialogs"></div>
  <script type="module" src="/src/main.jsx"></script>
</body>
```

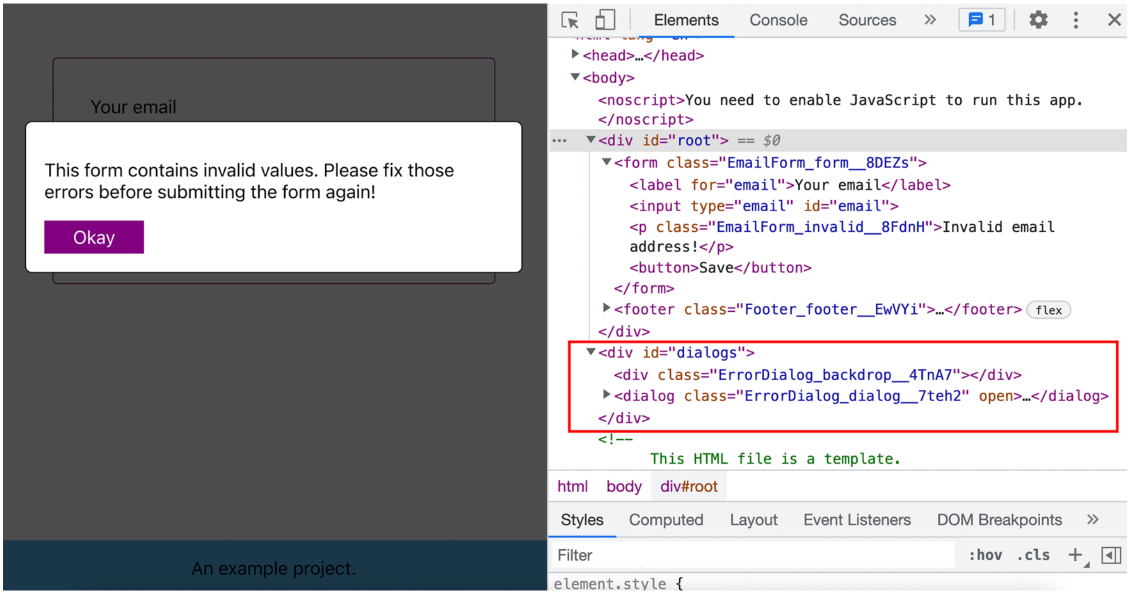
In this case, a `<div id="dialogs">` element is added in the `<body>` section, after the `<div id="root">` element to make sure that any components (and their styles) inserted in that element are evaluated last. This will ensure that their styles take a higher priority and overlay elements inserted into `<div id="dialogs">` would not be overlaid by other content coming earlier in the DOM. Adding and using multiple hooks would be possible, but for this example, only one *injection point* is needed. You can also use HTML elements other than `<div>` elements.

With the `index.html` file adjusted, React can be instructed to render certain JSX elements (i.e., components) in a specified *injection point* via the `createPortal()` function of `react-dom`:

```
import { createPortal } from 'react-dom';

import classes from './ErrorDialog.module.css';

function ErrorDialog({ onClose }) {
  return createPortal(
    <>
      <div className={classes.backdrop}></div>
      <dialog className={classes.dialog} open>
        <p>
          This form contains invalid values. Please fix those errors before
          submitting the form again!
        </p>
        <button onClick={onClose}>Okay</button>
      </dialog>
    </>,
    document.getElementById('dialogs')
  );
}

export default ErrorDialog;
```

Inside this `ErrorDialog` component, which is rendered conditionally by another component (the `EmailForm` component, the example code for which is available on GitHub), the returned JSX code is wrapped by `createPortal()`. `createPortal()` takes two arguments: the JSX code that should be rendered in the DOM and a pointer at the element in `index.html` where the content should be injected.

In this example, the newly added `<div id="dialogs">` is selected via `document.getElementById('dialogs')`. Therefore, `createPortal()` ensures that the JSX code generated by `ErrorDialog` is rendered in that place in the HTML document:



This form contains invalid values. Please fix those errors before submitting the form again!

```
▶<head>…</head>
▼<body>
   <noscript>You need to enable JavaScript to run this app.
   </noscript>
…  ▼<div id="root"> == $0
     ▼<form class="EmailForm_form__8DEZs">
        <label for="email">Your email</label>
        <input type="email" id="email">
        <p class="EmailForm_invalid__8FdnH">Invalid email
        address!</p>
        <button>Save</button>
      </form>
     ▶<footer class="Footer_footer__EwVYi">…</footer> flex
   </div>
   ▼<div id="dialogs">
      <div class="ErrorDialog_backdrop__4TnA7"></div>
     ▶<dialog class="ErrorDialog_dialog__7teh2" open>…</dialog>
   </div>
   <!--
         This HTML file is a template.
```

html body div#root

element.style {

Figure 7.8: The overlay elements are inserted into <div id="dialogs">

In this screenshot, you can see that the overlay elements (`<div>` backdrop and `<dialog>` error) are indeed inserted into the `<div id="dialogs">` element, instead of the `<form>` element (as they were before).

As a result of this change, `<footer>` no longer overlays the error dialog backdrop without any CSS code changes. Semantically, the final DOM structure also makes more sense since you would typically expect overlay elements to be closer to the root DOM node.

Still, using this portal feature is optional. The same visual result (though not the DOM structure) could have been achieved by changing some CSS styles. Nonetheless, aiming for a clean DOM structure is a worthwhile pursuit, and avoiding unnecessarily complex CSS code is also not a bad thing.

Summary and Key Takeaways

- Refs can be used to gain direct access to DOM elements or to store values that won't be reset or changed when the surrounding component is re-evaluated.
- Only use this direct access to read values, not to manipulate DOM elements (let React do this instead).

- Components that gain DOM access via Refs, instead of state and other React features, are considered uncontrolled components (because React is not in direct control).
- Prefer controlled components (using state and a strictly declarative approach) over uncontrolled components unless you're performing very simple tasks such as reading an entered input value.
- Using Refs, you can also expose features of your own components so that they may be used imperatively.
- You can set and use a `ref` prop on custom components when working with React 19 or higher.
- When using React < 19, React's `forwardRef()` function must be used to receive Refs on custom components.
- Portals can be used to instruct React to render JSX elements in a different place in the DOM than they normally would.

What's Next?

At this point in the book, you've encountered many key tools and concepts that can be used to build interactive and engaging user interfaces. Thanks to Refs, you can read DOM values without using state (hence avoiding unnecessary component re-evaluations) or manage values that persist across component updates. Thanks to Portals, you're able to control where exactly component markup is inserted into the DOM.

As a result, you get some new tools that can be used to fine-tune your React app. You may be able to improve performance (by avoiding component re-evaluations) or improve the structure and semantics of your DOM elements. Ultimately, it's the combination of all these tools that allows you to build engaging, interactive, and also performant web applications with React.

But, as you will learn in the next chapter, React has even more helpful core concepts to offer: for example, a way of handling **side effects**.

The next chapter will explore what exactly **side effects** are, why they need special handling, and how React helps you with that.

Test Your Knowledge!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers with examples that can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/07-portals-refs/exercises/questions-answers.md`.

1. How can Refs help with handling user input in forms?
2. What is an uncontrolled component?
3. What is a controlled component?
4. When should you **not** use Refs?
5. What's the main idea behind portals?

Apply What You Have Learned

With this newly gained knowledge about Refs and Portals, it's again time to practice what you have learned.

Below, you'll find two activities that allow you to practice working with Refs and Portals. As always, you will, of course, also need some of the concepts covered in earlier chapters (e.g., working with state).

Activity 7.1: Extract User Input Values

In this activity, you have to add logic to an existing React component to extract values from a form. The form contains an input field and a drop-down menu and you should make sure that, upon form submission, both values are read and, for the purpose of this dummy app, output to the browser console.

Use your knowledge about Refs and uncontrolled components to implement a solution without using React state.

> **Note**
>
> You can find the starting code for this activity at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/07-portals-refs/activities/practice-1-start`. When downloading this code, you'll always download the entire repository. Make sure to then navigate to the subfolder with the starting code (`activities/practice-1-start` in this case) to use the right code snapshot.

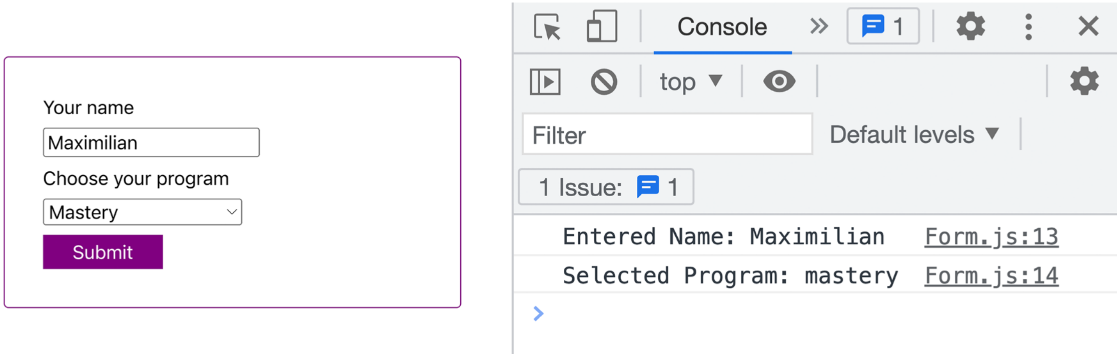
After downloading the code and running `npm install` in the project folder (to install all the required dependencies), the solution steps are as follows:

1. Create two Refs, one for each input element that should be read (input field and drop-down menu).
2. Connect the Refs to the input elements.
3. In the submit handler function, access the connected DOM elements via the Refs and read the currently entered or selected values.
4. Output the values to the browser console.

Book Code for Supplementary Content

Your book code for the supplementary content package included with this book is **L58S0LOR5U**. To unlock the content, which includes per-chapter cheatsheets and videos from the author, follow the instructions here: `https://packt.link/supplementary-content-9781836202271`.

The expected result (user interface) should look like this:



Your name
Maximilian
Choose your program
Mastery

```
Entered Name: Maximilian      Form.js:13
Selected Program: mastery     Form.js:14
```

Figure 7.9: The browser developer tools console outputs the selected values

> Note
>
> You will find all code files used for this activity, as well as the solution, at https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/07-portals-refs/activities/practice-1.

Activity 7.2: Add a Side Drawer

In this activity, you will connect an already existing `SideDrawer` component with a button in the main navigation bar to open the side drawer (i.e., display it) whenever the button is clicked. After the side drawer opens, a click on the backdrop should close the drawer again.

In addition to implementing the general logic described above, your goal will be to ensure proper positioning in the final DOM so that no other elements are overlaid on top of the `SideDrawer` (without editing any CSS code). The `SideDrawer` should also not be nested in any other components or JSX elements.

> **Note**
>
> This activity comes with some starting code, which can be found here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/07-portals-refs/activities/practice-2-start`.

After downloading the code and running `npm install` to install all the required dependencies, the solution steps are as follows:

1. Add logic to conditionally show or hide the `SideDrawer` component in the `MainNavigation` component.
2. Add an *injection hook* for the side drawer in the HTML document.
3. Use React's portal feature to render the JSX elements of `SideDrawer` in the newly added hook.

The final user interface should look and behave like this:

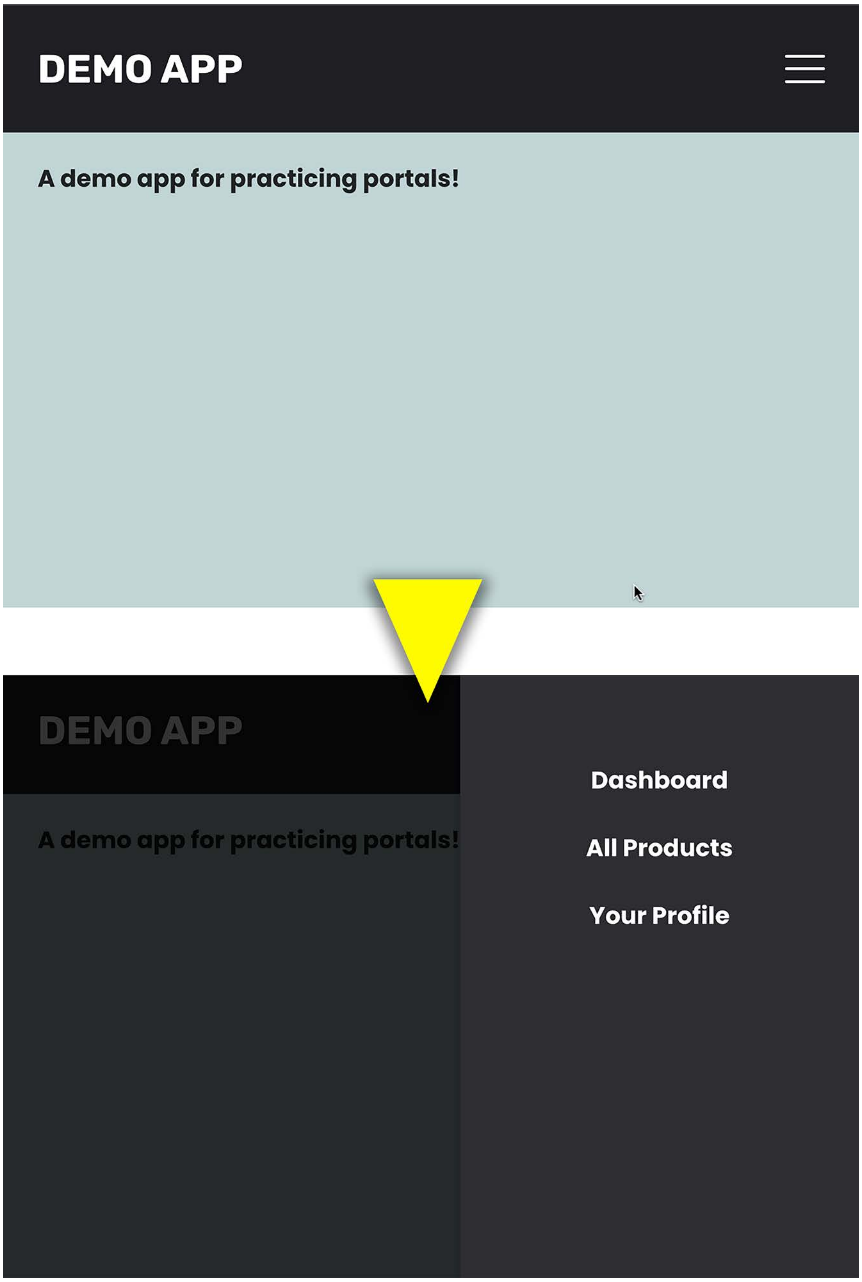
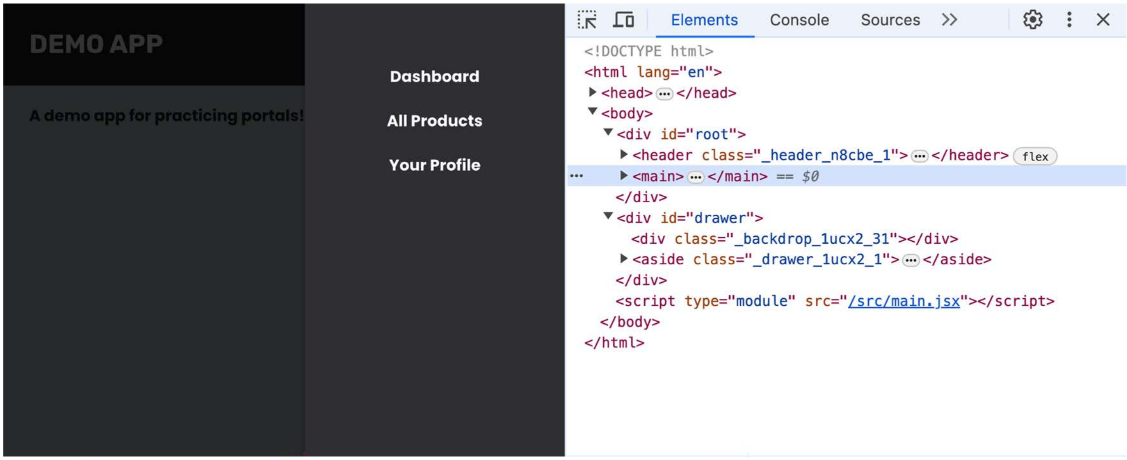

DEMO APP
A demo app for practicing portals!
DEMO APP
A demo app for practicing portals!
Dashboard
All Products
Your Profile

Figure 7.10: A click on the menu button opens the side drawer

Upon clicking on the menu button, the side drawer opens. If the backdrop behind the side drawer is clicked, it should close again.

The final DOM structure (with the side drawer opened) should look like this:


DEMO APP
A demo app for practicing portals!
Dashboard
All Products
Your Profile
Elements
Console
Sources

```
<!DOCTYPE html>
<html lang="en">
 ▶<head> ··· </head>
 ▼<body>
   ▼<div id="root">
     ▶<header class="_header_n8cbe_1"> ··· </header> flex
     ▶<main> ··· </main> == $0
    </div>
   ▼<div id="drawer">
      <div class="_backdrop_1ucx2_31"></div>
     ▶<aside class="_drawer_1ucx2_1"> ··· </aside>
    </div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

Figure 7.11: The drawer-related elements are inserted in a separate place in the DOM

The side drawer-related DOM elements (the `<div>` backdrop and `<aside>`) are inserted into a separate DOM node (`<div id="drawer">`).

Note

You will find all code files used for this activity, as well as the solution, at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/07-portals-refs/activities/practice-2`.

8

Handling Side Effects

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Identify side effects in your React apps
- Understand and use the `useEffect()` Hook
- Utilize the different features and concepts related to the `useEffect()` Hook to avoid bugs and optimize your code
- Handle side effects related and unrelated to state changes

Introduction

While all React examples previously covered in this book have been relatively straightforward, and many key React concepts were introduced, it is unlikely that many real apps could be built with those concepts alone.

Most real apps that you will build as a React developer also need to send HTTP requests, access the browser storage and log analytics data, or perform any other kind of similar task, and with components, props, events, and state alone, you'll often encounter problems when trying to add such features to your app. Detailed explanations and examples will be discussed later in this chapter, but the core problem is that tasks like this will often interfere with React's component rendering cycle, leading to unexpected bugs or even breaking the app.

This chapter will take a closer look at those kinds of actions, analyze what they have in common, and most importantly, teach you how to correctly handle such tasks in React apps.

What's the Problem?

Before exploring a solution, it's important to first understand the concrete problem.

Actions that are not directly related to producing a (new) user interface state often clash with React's component rendering cycle. They may introduce bugs or even break the entire web app.

Consider the following example code snippet (important: don't execute this code as it will cause an infinite loop and send a large number of HTTP requests behind the scenes):

```
import { useState } from 'react';

import classes from './BlogPosts.module.css';

async function fetchPosts() {
  const response = await fetch('https://jsonplaceholder.typicode.com/posts');
  const blogPosts = await response.json();
  return blogPosts;
}

function BlogPosts() {
  const [loadedPosts, setLoadedPosts] = useState([]);

  fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));

  return (
    <ul className={classes.posts}>
      {loadedPosts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

export default BlogPosts;
```

So what's the problem with this code? Why does it create an infinite loop?

In this example, a React component (`BlogPosts`) is created. In addition, a non-component function (`fetchPosts()`) is defined. That function uses the built-in `fetch()` function (provided by the browser) to send an HTTP request to an external **application programming interface** (**API**) and fetch some data.

Note

The `fetch()` function is made available by the browser (all modern browsers support this function). You can learn more about `fetch()` at `https://academind.com/tutorials/xhr-fetch-axios-the-fetch-api`.

The `fetch()` function yields a **promise**, which, in this example, is handled via `async/await`. Just like `fetch()`, promises are a key web development concept, which you can learn more about (along with `async/await`) at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function`.

An API, in this context, is a site that exposes various paths to which requests can be sent—either to submit or to fetch data. `jsonplaceholder.typicode.com` is a dummy API, responding with dummy data. It can be used in scenarios like the preceding example, where you just need an API to send requests to. You can use it to test some concept or code without connecting or creating a real backend API. In this case, it's used to explore some React problems and concepts. Basic knowledge about sending HTTP requests with `fetch()` and APIs is expected for this chapter and the book overall. If needed, you can use pages such as MDN (`https://developer.mozilla.org/`) to strengthen your knowledge of such core concepts.

In the preceding code snippet, the `BlogPosts` component utilizes `useState()` to register a `loadedPosts` state value. The state is used to output a list of blog posts. Those blog posts are not defined in the app itself though. Instead, they are fetched from the external API mentioned in the note box.

`fetchPosts()`, which is the utility function that contains the code for fetching blog posts data from that backend API using the built-in `fetch()` function, is called directly in the component function body. Since `fetchPosts()` is an `async` function (using `async/await`), it returns a promise. In `BlogPosts`, the code that should be executed once the promise resolves is registered via the built-in `then()` method.

Note

`async/await` is not used directly in the component function body because regular React components must not be `async` functions. Such functions automatically return a promise as a value (even without an explicit `return` statement), which is an invalid return value for a React component.

That being said, there are indeed React components that are allowed to use `async/await` and return a promise. So-called **React Server Components** are not restricted to returning JSX code, strings, etc. This feature will be discussed in detail in *Chapter 16*, *React Server Components & Server Actions*.

Once the `fetchPosts()` promise resolves, the extracted posts data (`fetchedPosts`) is set as the new `loadedPosts` state (via `setLoadedPosts(fetchedPosts)`).

If you were to run the preceding code (which you should not do!), it would at first seem to work. But behind the scenes, it would actually start an infinite loop, hammering the API with HTTP requests. This is because, as a result of getting a response from the HTTP request, `setLoadedPosts()` is used to set a new state.

Earlier in this book (in *Chapter 4*, *Working with Events and State*), you learned that whenever the state of a component changes, React re-evaluates the component to which the state belongs. "Re-evaluating" simply means that the component function is executed again (by React, automatically).

Since this `BlogPosts` component calls `fetchPosts()` (which sends an HTTP request) directly inside the component function body, this HTTP request will be sent every time the component function is executed. And as the state (`loadedPosts`) is updated as a result of getting a response from that HTTP request, this process begins again, and an infinite loop is created.

The root problem, in this case, is that sending an HTTP request is a side effect—a concept that will be explored in greater detail in the next section.

Understanding Side Effects

Side effects are actions or processes that occur in addition to another *main process*. At least, this is a concise definition that helps with understanding side effects in the context of a React app.

Note

If you want to dive deeper into the concept of *side effects*, you can also explore the following discussion about side effects on Stack Overflow: `https://softwareengineering.stackexchange.com/questions/40297/what-is-a-side-effect`.

In the case of a React component, the main process would be the component render cycle in which the main task of a component is to render the user interface that is defined in the component function (the returned JSX code). The React component should return the final JSX code, which is then translated into DOM-manipulating instructions.

For this, React considers state changes as the trigger for updating the user interface. Registering event handlers such as `onClick`, adding refs, or rendering child components (possibly by using props) would be other elements that belong to this main process—because all these concepts are directly related to the main task of rendering the desired user interface.

Sending an HTTP request, as in the preceding example, is not part of this main process, though. It doesn't directly influence the user interface. While the response data might eventually be output on the screen, it definitely won't be used in the exact same component render cycle in which the request is sent (because HTTP requests are asynchronous tasks).

Since sending the HTTP request is not part of the main process (rendering the user interface) that's performed by the component function, it's considered a side effect. It's invoked by the same function (the `BlogPosts` component function), which primarily has a different goal.

If the HTTP request were sent upon a click of a button rather than as part of the main component function body, it would not be a side effect. Consider this example:

```
import { useState } from 'react';

import classes from './BlogPosts.module.css';

async function fetchPosts() {
  const response = await fetch('https://jsonplaceholder.typicode.com/posts');
  const blogPosts = await response.json();
  return blogPosts;
}

function BlogPosts() {
  const [loadedPosts, setLoadedPosts] = useState([]);

  function handleFetchPosts() {
    fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
  }

  return (
    <>
      <button onClick={handleFetchPosts}>Fetch Posts</button>
      <ul className={classes.posts}>
        {loadedPosts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </>
  );
}

export default BlogPosts;
```

This code is almost identical to the previous example, but it has one important difference: a `<button>` was added to the JSX code. And it's this button that invokes a newly added `handleFetchPosts()` function, which then sends the HTTP request (and updates the state).

With this change made, the HTTP request is *not* sent every time the component function re-renders (that is, is executed again). Instead, it's only sent whenever the button is clicked, and therefore, this does not create an infinite loop. The HTTP request, in this case, also doesn't postulate a side effect, because the primary goal of `handleFetchPosts()` (i.e., the main process) is to fetch new posts and update the state.

Side Effects Are Not Just about HTTP Requests

In the previous example, you learned about one potential side effect that could occur in a component function: an HTTP request. You also learned that HTTP requests are not always side effects. It depends on where they are created.

In general, any action that's started upon the execution of a React component function is a side effect if that action is not directly related to the main task of rendering the component's user interface.

Here's a non-exhaustive list of examples of side effects:

- Sending an HTTP request (as shown previously)
- Storing data to or fetching data from browser storage (for example, via the built-in `localStorage` object)
- Setting timers (via `setTimeout()`) or intervals (via `setInterval()`)
- Logging data to the console via `console.log()`

Not all side effects cause infinite loops, however. Such loops only occur if the side effect leads to a state update.

Here's an example of a side effect that would not cause an infinite loop:

```
function ControlCenter() {
  function handleStart() {
    // do something ...
  }

  console.log('Component is rendering!'); // this is a side effect!

  return (
    <div>
      <p>Press button to start the review process</p>
      <button onClick={handleStart}>Start</button>
    </div>
  );
}
```

In this example, `console.log(…)` is a side effect because it's executed as part of every component function execution and does not influence the rendered user interface (neither for this specific render cycle nor indirectly for any future render cycles in this case, unlike the previous example with the HTTP request).

Of course, using `console.log()` like this is not causing any problems. During development, it's quite normal to log messages or data for debugging purposes. Side effects aren't necessarily a problem and, indeed, side effects like this can be used or tolerated.

But you also often need to deal with side effects such as the HTTP request from before. Sometimes, you need to fetch data when a component renders—probably not for every render cycle, but typically the first time it is executed (that is, when its generated user interface appears on the screen for the first time).

React offers a solution for this kind of problem as well.

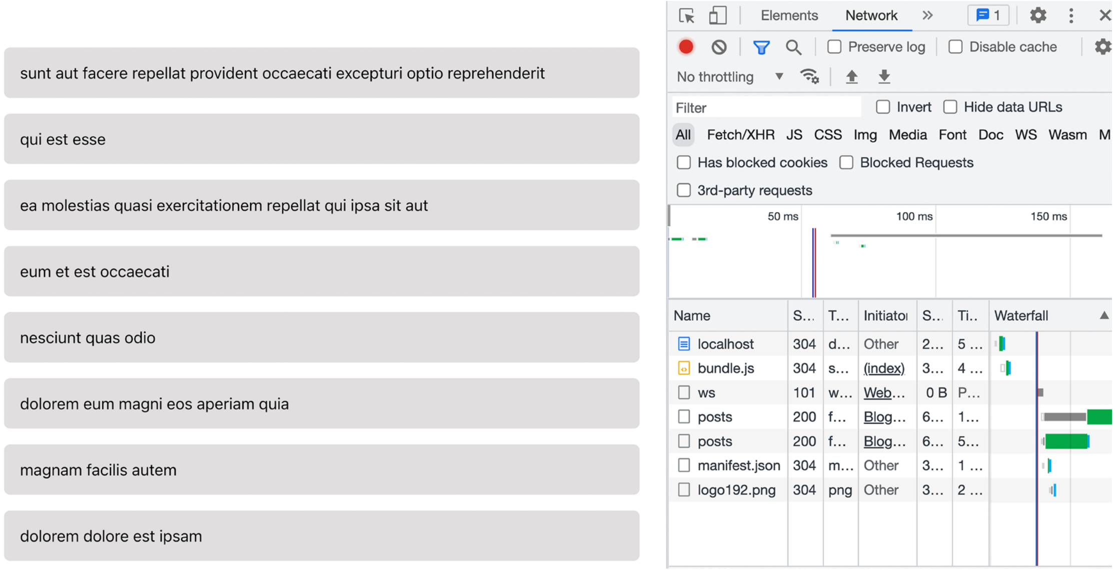
Dealing with Side Effects with the useEffect() Hook

In order to deal with side effects such as the HTTP request shown previously in a safe way (that is, without creating an infinite loop), React offers another core Hook: the `useEffect()` Hook.

The first example can be fixed and rewritten like this:

```
import { useState, useEffect } from 'react';

import classes from './BlogPosts.module.css';

async function fetchPosts() {
  const response = await fetch('https://jsonplaceholder.typicode.com/posts');
  const blogPosts = await response.json();
  return blogPosts;
}

function BlogPosts() {
  const [loadedPosts, setLoadedPosts] = useState([]);

  useEffect(function () {
    fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
  }, []);

  return (
    <ul className={classes.posts}>
      {loadedPosts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

export default BlogPosts;
```

In this example, the `useEffect()` Hook is imported and used to control the side effect (hence the name of the Hook, `useEffect()`, as it deals with side effects in React components). The exact syntax and usage will be explored in the next section, but if you use this Hook, you can safely run the example and get some output like this:


sunt aut facere repellat provident occaecati excepturi optio reprehenderit

qui est esse

ea molestias quasi exercitationem repellat qui ipsa sit aut

eum et est occaecati

nesciunt quas odio

dolorem eum magni eos aperiam quia

magnam facilis autem

dolorem dolore est ipsam

Name	S...	T...	Initiator	S...	Ti...	Waterfall
localhost	304	d...	Other	2...	5 ...	
bundle.js	304	s...	(index)	3...	4 ...	
ws	101	w...	Web...	0 B	P...	
posts	200	f...	Blog...	6...	1...	
posts	200	f...	Blog...	6...	5...	
manifest.json	304	m...	Other	3...	1 ...	
logo192.png	304	png	Other	3...	2 ...	

Figure 8.1: A list of dummy blog posts and no infinite loop of HTTP requests

In the preceding screenshot, you can see the list of dummy blog post titles, and most importantly, when inspecting the sent network requests, you find no infinite list of requests.

`useEffect()` is therefore the solution for problems like the one outlined previously. It helps you deal with side effects so that you can avoid infinite loops and extract them from your component function's main process.

But how does `useEffect()` work, and how is it used correctly?

How to Use useEffect()

As shown in the previous example code snippet, `useEffect()`, like all React Hooks, is executed as a function inside the component function (`BlogPosts`, in this case).

Although, unlike `useState()` or `useRef()`, `useEffect()` does not return a value, though it does accept an argument (or, actually, two arguments) like those other Hooks. The first argument is *always* a function. In this case, the function passed to `useEffect()` is an anonymous function, created via the `function` keyword.

Alternatively, you could also provide an anonymous function created as an arrow function (`useEffect(() => { … })`) or point at some named function (`useEffect(doSomething)`). The only thing that matters is that the first argument passed to `useEffect()` *must* be a function. It must not be any other kind of value.

In the preceding example, `useEffect()` also receives a second argument: an empty array (`[]`). The second argument must be an array, but providing it is *optional*. You could also omit the second argument and just pass the first argument (the function) to `useEffect()`. However, in most cases, the second argument is needed to achieve the correct behavior. Both arguments and their purpose will be explored in greater detail as follows.

The first argument is a function that will be executed by React. It will be executed *after* every component render cycle (that is, after every component function execution).

In the preceding example, if you only provide this first argument and omit the second, you will therefore still create an infinite loop. There will be an (invisible) timing difference because the HTTP request will now be sent after every component function execution (instead of as part of it), but you will still trigger a state change, which will still trigger the component function to execute again. Therefore, the effect function will run again, and an infinite loop will be created. In this case, the side effect will be extracted out of the component function technically, but the problem with the infinite loop will not be solved:

```
useEffect(function () {
  fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
}); // this would cause an infinite loop again!
```

Extracting side effects out of React component functions is the main job of `useEffect()`, and so only the first argument (the function that contains the side effect code) is mandatory. But, as mentioned previously, you will also typically need the second argument to control the frequency with which the effect code will be executed, because that's what the second argument (an array) will do.

The second parameter received by `useEffect()` is *always* an array (unless it's omitted). This array specifies the dependencies of the effect function. Any dependency specified in this array will, once it changes, cause the effect function to execute again. If no array is specified (that is, if the second argument is omitted), the effect function will be executed again for every component function execution:

```
useEffect(function () {
  fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
}, []);
```

In the preceding example, the second argument was not omitted, but it's an empty array. This informs React that this effect function has no dependencies. Therefore, the effect function will never be executed again. Instead, it will only be executed once, when the component is rendered for the first time. If you set no dependencies (by providing an empty array), React will execute the effect function *once*—directly after the component function was executed for the first time.

It's important to note that specifying an empty array is very different from omitting it. If it is omitted, no dependency information is provided to React. Therefore, React executes the effect function after every component re-evaluation. If an empty array is provided instead, you explicitly state that this effect has no dependencies and therefore should only run once.

This brings up another important question, though: when should you add dependencies? And how exactly are dependencies added or specified?

Effects and Their Dependencies

Omitting the second argument to `useEffect()` causes the effect function (the first argument) to execute after every component function execution. Providing an empty array causes the effect function to run only once (after the first component function invocation). But is that all you can control?

No, it isn't.

The array passed to `useEffect()` can and should contain all variables, constants, or functions that are used inside the effect function—if those variables, constants, or functions were defined inside the component function (or in some parent component function, passed down via props).

Consider this example:

```
import { useState, useEffect } from 'react';

import classes from './BlogPosts.module.css';

async function fetchPosts(url) {
  const response = await fetch(url);
  const blogPosts = await response.json();
  return blogPosts;
}

function BlogPosts({ url }) {
  const [loadedPosts, setLoadedPosts] = useState([]);

  useEffect(function () {
    fetchPosts(url)
     .then((fetchedPosts) => setLoadedPosts(fetchedPosts));
  }, [url]);

  return (
    <ul className={classes.posts}>
      {loadedPosts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

export default BlogPosts;
```

This example is based on the previous example, but was adjusted in one important place: `BlogPosts` now accepts a `url` prop.

Therefore, this component can now be used and configured by other components. Of course, if some other component sets a URL that doesn't return a list of blog posts, the app won't work as intended. This component therefore might be of limited practical use, but it does show the importance of effect dependencies quite well.

But if that other component changes the URL (e.g., due to some user input there), a new request should be sent, of course. So `BlogPosts` should send another fetch request every time the `url` prop value changes.

That's why `url` was added to the dependencies array of `useEffect()`. If the array had been kept empty, the effect function would only run once (as described in the previous section). Therefore, any changes to `url` wouldn't have any effect (no pun intended) on the effect function or the HTTP request executed as part of that function. No new HTTP request would be sent.

By adding `url` to the dependencies array, React registers this value (in this case, a prop value, but any value can be registered) and re-executes the effect function whenever that value changes (that is, whenever a new `url` prop value is set by the component that uses `BlogPosts`).

The most common types of effect dependencies are state values, props, and functions that might be executed inside of the effect function. The latter will be analyzed in greater depth later in this chapter.

As a rule, you should add all values (including functions) that are used inside an effect function to the effect dependencies array.

With this new knowledge in mind, if you take another look at the preceding `useEffect()` example code, you might spot some missing dependencies:

```
useEffect(function () {
  fetchPosts(url)
    .then((fetchedPosts) => setLoadedPosts(fetchedPosts));
}, [url]);
```

Why are `fetchPosts`, `fetchedPosts`, and `setLoadedPosts` not added as dependencies? These are, after all, values and functions used inside of the effect function. The next section will address this in detail.

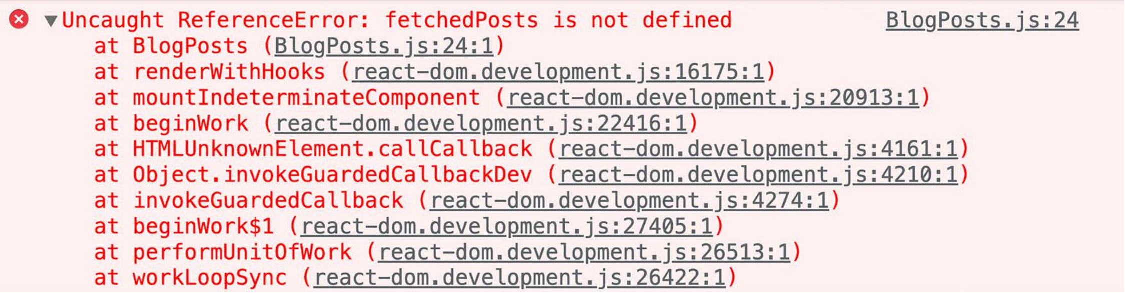
Unnecessary Dependencies

In the previous example, it might seem as if `fetchPosts`, `fetchedPosts`, and `setLoadedPosts` should be added as dependencies to `useEffect()`, as shown here:

```
useEffect(function () {
  fetchPosts(url)
    .then((fetchedPosts) => setLoadedPosts(fetchedPosts));
}, [url, fetchPosts, fetchedPosts, setLoadedPosts]);
```

However, for `fetchPosts` and `fetchedPosts`, this would be incorrect. And for `setLoadedPosts`, it would be unnecessary.

`fetchedPosts` should not be added because it's not an external dependency. It's a local variable (or argument, to be precise), defined and used inside the effect function. It's not defined in the component function to which the effect belongs. If you try to add it as a dependency, you'll get an error:



```
▼Uncaught ReferenceError: fetchedPosts is not defined          BlogPosts.js:24
    at BlogPosts (BlogPosts.js:24:1)
    at renderWithHooks (react-dom.development.js:16175:1)
    at mountIndeterminateComponent (react-dom.development.js:20913:1)
    at beginWork (react-dom.development.js:22416:1)
    at HTMLUnknownElement.callCallback (react-dom.development.js:4161:1)
    at Object.invokeGuardedCallbackDev (react-dom.development.js:4210:1)
    at invokeGuardedCallback (react-dom.development.js:4274:1)
    at beginWork$1 (react-dom.development.js:27405:1)
    at performUnitOfWork (react-dom.development.js:26513:1)
    at workLoopSync (react-dom.development.js:26422:1)
```

Figure 8.2: An error occurred—fetchedPosts could not be found

`fetchPosts`, the function that sends the actual HTTP request, is not a function defined inside of the effect function. But it still shouldn't be added because it is defined outside the component function.

Therefore, there is no way for this function to change. It's defined once (in the `BlogPosts.jsx` file), and it can't change. That said, this would not be the case if it were defined inside the component function. In that case, whenever the component function executes again, the `fetchPosts` function would be recreated as well. This is a scenario that will be discussed later in this chapter (in the *Functions as Dependencies* section).

In this example though, `fetchPosts` can't change. Therefore, it doesn't have to be added as a dependency (and consequently should not be). The same would be true for functions, or any kind of values, provided by the browser or third-party packages. Any value that's not defined inside a component function shouldn't be added to the dependencies array.

Note

It may be confusing that a function could change—after all, the logic is hardcoded, right? But in JavaScript, functions are actually just objects and therefore may change. When the code that contains a function is executed again (e.g., a component function being executed again by React), a new function object will be created in memory.

If this is not something you're familiar with, the following resource should be helpful: `https://academind.com/tutorials/javascript-functions-are-objects`.

So `fetchedPosts` and `fetchPosts` should both not be added (for different reasons). What about `setLoadedPosts`?

`setLoadedPosts` is the state updating function returned by `useState()` for the `loadedPosts` state value. Therefore, like `fetchPosts`, it's a function. Unlike `fetchPosts`, though, it's a function that's defined inside the component function (because `useState()` is called inside the component function). It's a function created by React (since it's returned by `useState()`), but it's still a function. Theoretically, it should therefore be added as a dependency. And indeed, you can add it without any negative consequences.

But state updating functions returned by `useState()` are a special case: React guarantees that those functions will never change or be recreated. When the surrounding component function (`BlogPosts`) is executed again, `useState()` also executes again. However, a new state updating function is only created the first time a component function is called by React. Subsequent executions don't lead to a new state updating function being created.

Because of this special behavior (i.e., React guaranteeing that the function itself never changes), state updating functions may (and actually should) be omitted from the dependencies array.

For all these reasons, `fetchedPosts`, `fetchPosts`, and `setLoadedPosts` should all not be added to the dependencies array of `useEffect()`. `url` is the only dependency used by the effect function that may change (that is, when the user enters a new URL into the input field) and therefore should be listed in the array.

To sum it up, when it comes to adding values to the effect dependencies array, there are three kinds of exceptions:

- Internal values (or functions) that are defined and used inside the effect (such as `fetchedPosts`)
- External values that are not defined inside a component function (such as `fetchPosts`)
- State updating functions (such as `setLoadedPosts`)

In all other cases, if a value is used in the effect function, it *must be added* to the dependencies array! Omitting values incorrectly can lead to unexpected effect executions (that is, an effect executing too often or not often enough).

Cleaning Up after Effects

To perform a certain task (for example, sending an HTTP request), many effects should simply be triggered when their dependencies change. While some effects can be re-executed multiple times without issue, there are also effects that, if they execute again before the previous task has finished, are an indication that the task performed needs to be canceled. Or, maybe there is some other kind of cleanup work that should be performed when the same effect executes again.

Here's an example, where an effect sets a timer:

```
import { useState, useEffect } from 'react';

function Alert() {
  const [alertDone, setAlertDone] = useState(false);

  useEffect(function () {
    console.log('Starting Alert Timer!');
    setTimeout(function () {
      console.log('Timer expired!');
      setAlertDone(true);
    }, 2000);
  }, []);
  return (
    <>
      {!alertDone && <p>Relax, you still got some time!</p>}
      {alertDone && <p>Time to get up!</p>}
    </>
  );
}

export default Alert;
```

This `Alert` component is used in the `App` component:

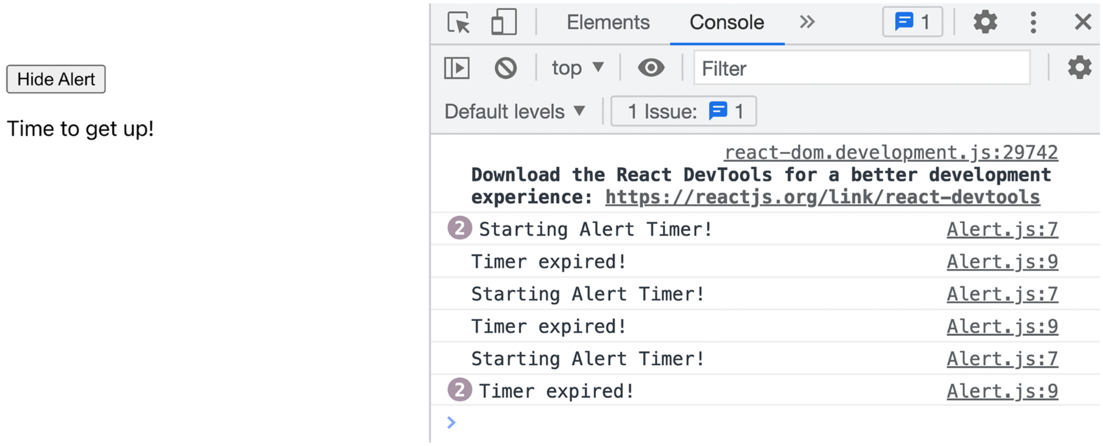
```
import { useState } from 'react';

import Alert from './components/Alert.jsx';

function App() {
  const [showAlert, setShowAlert] = useState(false);

  function handleShowAlert() {
    // state updating is done by passing a function to setShowAlert
    // because the new state depends on the previous state (it's the opposite)
    setShowAlert((isShowing) => !isShowing);
  }

  return (
    <>
      <button onClick={handleShowAlert}>
        {showAlert ? 'Hide' : 'Show'} Alert
      </button>
      {showAlert && <Alert />}
    </>
  );
}

export default App;
```

In the `App` component, the `Alert` component is shown conditionally. The `showAlert` state is toggled via the `handleShowAlert` function (which is triggered upon a button click).

In the `Alert` component, a timer is set using `useEffect()`. Without `useEffect()`, an infinite loop would be created, since the timer, upon expiration, changes some component state (the `alertDone` state via the `setAlertDone` state updating function).

The dependency array is an empty array because this effect function does not use any component values, variables, or functions. `console.log()` and `setTimeout()` are functions built into the browser (and therefore external functions), and `setAlertDone()` can be omitted because of the reasons mentioned in the previous section.

If you run this app and then start toggling the alert (by clicking the button), you'll notice strange behavior. The timer is set every time the `Alert` component is rendered. But it's not clearing the existing timer. This is due to the fact that multiple timers are running simultaneously, as you can clearly see if you look at the JavaScript console in your browser's developer tools:



```
react-dom.development.js:29742
Download the React DevTools for a better development
experience: https://reactjs.org/link/react-devtools
2 Starting Alert Timer!        Alert.js:7
Timer expired!                 Alert.js:9
Starting Alert Timer!          Alert.js:7
Timer expired!                 Alert.js:9
Starting Alert Timer!          Alert.js:7
2 Timer expired!               Alert.js:9
```

Figure 8.3: Multiple timers are started

This example is deliberately kept simple, but there are other scenarios in which you may have an ongoing HTTP request that should be aborted before a new one is sent. There are cases like that, where an effect should be cleaned up first before it runs again.

React also provides a solution for those kinds of situations: the effect function passed as a first argument to `useEffect()` can return an optional cleanup function. If you do return a function inside your effect function, React will execute that function every time *before* it runs the effect again.

Here's the `useEffect()` call of the `Alert` component with a cleanup function being returned:

```
useEffect(function () {
  let timer;

  console.log('Starting Alert Timer!');
  timer = setTimeout(function () {
    console.log('Timer expired!');
    setAlertDone(true);
  }, 2000);

  return function() {
    clearTimeout(timer);
  }
}, []);
```

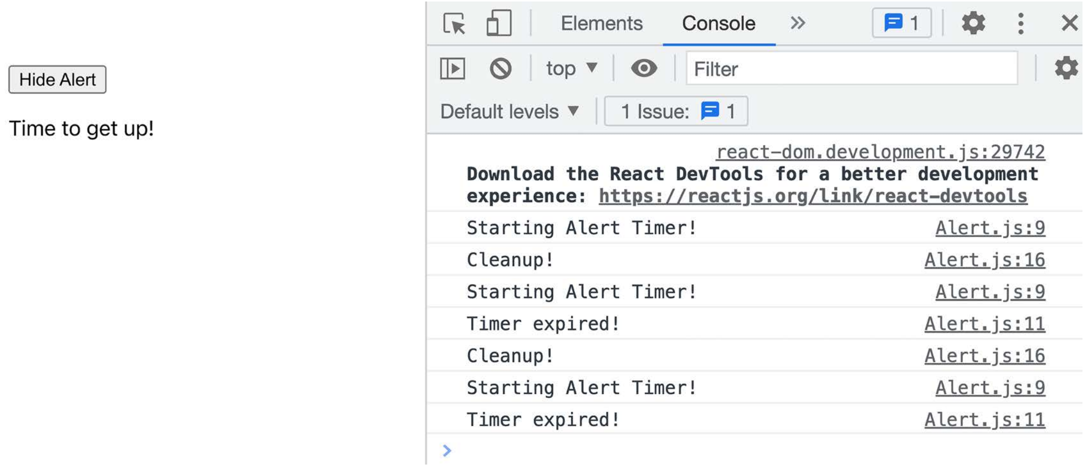
In this updated example, a new `timer` variable (a local variable that is only accessible inside the effect function) is added. That variable stores a reference to the timer that's created by `setTimeout()`. This reference can then be used together with `clearTimeout()` to remove a timer.

The timer is removed in a function returned by the effect function—which is the cleanup function that will be executed automatically by React before the effect function is called the next time.

You can see the cleanup function in action if you add a `console.log()` statement to it:

```
return function() {
  console.log('Cleanup!');
  clearTimeout(timer);
}
```

In your JavaScript console, this looks as follows:



Time to get up!

```
                                 react-dom.development.js:29742
Download the React DevTools for a better development
experience: https://reactjs.org/link/react-devtools
Starting Alert Timer!                                Alert.js:9
Cleanup!                                            Alert.js:16
Starting Alert Timer!                                Alert.js:9
Timer expired!                                      Alert.js:11
Cleanup!                                            Alert.js:16
Starting Alert Timer!                                Alert.js:9
Timer expired!                                      Alert.js:11
>
```

Figure 8.4: The cleanup function is executed before the effect runs again

In the preceding screenshot, you can see that the cleanup function is executed (indicated by the `Cleanup!` log) right before the effect function is executed again. You can also see that the timer is cleared successfully: the first timer never expires (there is no `Timer expired!` log for the first timer in the screenshot).

The cleanup function is not executed when the effect function is called for the first time. However, it will be called by React whenever a component that contains an effect unmounts (that is, when it's removed from the DOM).

If an effect has multiple dependencies, the effect function will be executed whenever any of the dependency values change. Therefore, the cleanup function will also be called every time some dependency changes.

Dealing with Multiple Effects

Thus far, all the examples in this chapter have dealt with only one `useEffect()` call. You are not limited to only one call per component though. You can call `useEffect()` as often as needed—and can therefore register as many effect functions as needed.

But how many effect functions do you need?

You could start putting every side effect into its own `useEffect()` wrapper. You could put every HTTP request, every `console.log()` statement, and every timer into separate effect functions.

That said, as you can see in some of the previous examples—specifically, the code snippet in the previous section—that's not necessary. There, you have multiple effects in one `useEffect()` call (three `console.log()` statements and one timer).

A better approach would be to split your effect functions by dependencies. If one side effect depends on state A and another side effect depends on state B, you could put them into separate effect functions (unless those two states are related), as shown here:

```
function Demo() {
  const [a, setA] = useState(0); // state updating functions aren't called
  const [b, setB] = useState(0); // in this example

  useEffect(function() {
    console.log(a);
  }, [a]);

  useEffect(function() {
    console.log(b);
  }, [b]);

  // return some JSX code ...
}
```

But the best approach is to split your effect functions by logic. If one effect deals with fetching data via an HTTP request and another effect is about setting a timer, it will often make sense to put them into different effect functions (that is, different `useEffect()` calls).

Functions as Dependencies

Different effects have different kinds of dependencies, and one common kind of dependency is functions.

As mentioned previously, functions in JavaScript are just objects. Therefore, whenever some code that contains a function definition is executed, a new function object is created and stored in memory. When calling a function, it's that specific function object in memory that is executed. In some scenarios (for example, for functions defined in component functions), it's possible that multiple objects based on the same function code exist in memory.

Because of this behavior, functions that are referenced in code are not necessarily equal, even if they are based on the same function definition.

Consider this example:

```
function Alert() {
  function setAlert() {
    setTimeout(function() {
      console.log('Alert expired!');
    }, 2000);
  }

  useEffect(function() {
    setAlert();
  }, [setAlert]);

  // return some JSX code ...
}
```

In this example, instead of creating a timer directly inside the effect function, a separate `setAlert()` function is created in the component function. That `setAlert()` function is then used in the effect function passed to `useEffect()`. Since that function is used there, and because it's defined in the component function, it should be added as a dependency to `useEffect()`.

Another reason for this is that every time the `Alert` component function is executed again (e.g., because some state or prop value changes), a new `setAlert` function object will be created. In this example, that wouldn't be problematic because `setAlert` only contains static code. A new function object created for `setAlert` would work exactly in the same way as the previous one; therefore, it would not matter.

But now consider this adjusted example:

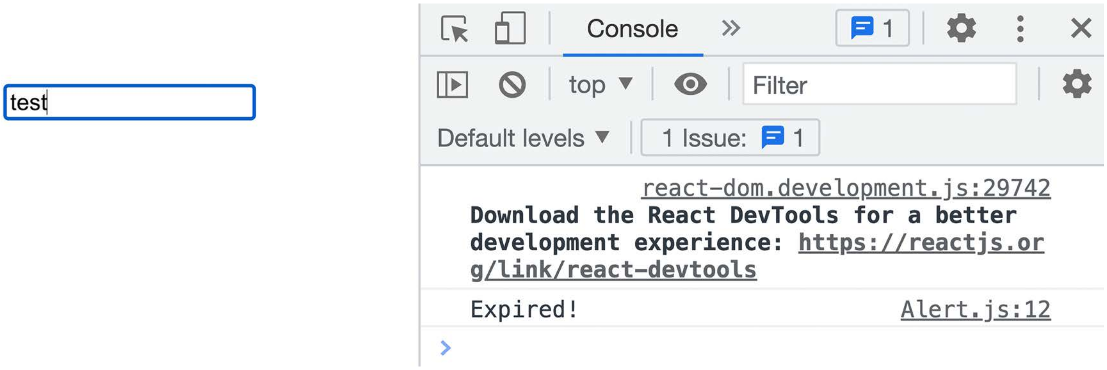
Note

The complete app can be found on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/08-effects/examples/function-dependencies`.

```
function Alert() {
  const [alertMsg, setAlertMsg] = useState('Expired!');

  function handleChangeAlertMsg(event) {
    setAlertMsg(event.target.value);
  }

  function setAlert() {
    setTimeout(function () {
        console.log(alertMsg);
      }, 2000);
    }

    useEffect(
      function () {
        setAlert();
      },
      []
    );

    return <input type="text" onChange={handleChangeAlertMsg} />;
  }

  export default Alert;
```

Now, a new `alertMsg` state is used for setting the actual alert message that's logged to the console. In addition, the `setAlert` dependency was removed from `useEffect()`.

If you run this code, you'll get the following output:



```
react-dom.development.js:29742
Download the React DevTools for a better
development experience: https://reactjs.or
g/link/react-devtools
Expired!                                    Alert.js:12
```

Figure 8.5: The console log does not reflect the entered value

In this screenshot, you can see that, despite a different value being entered into the input field, the original alert message is output.

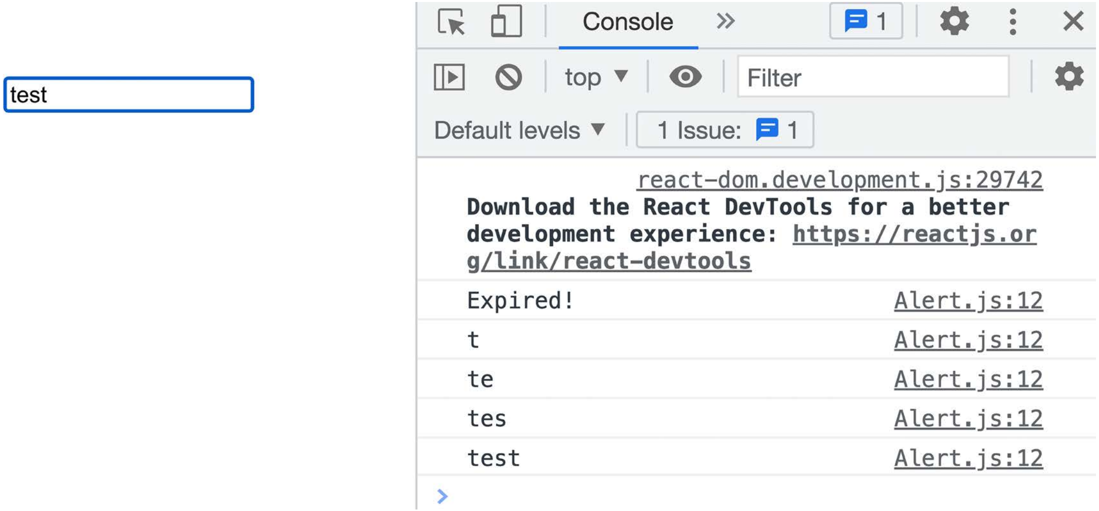
The reason for this behavior is that the new alert message is not picked up. It's not used because, despite the component function being executed again (because the state changed), the effect is not executed again. And the original execution of the effect still uses the old version of the `setAlert` function—the old `setAlert` function object, which has the old alert message locked in. That's how JavaScript functions work, and that's why, in this case, the desired result is not achieved.

The solution to the problem is simple though: add `setAlert` as a dependency to `useEffect()`. You should always add all values, variables, or functions used in an effect as dependencies, and this example shows *why* you should do that. Even functions can change.

If you add `setAlert` to the effect dependency array, you'll get a different output:

```
useEffect(
  function () {
    setAlert();
  },
  [setAlert]
);
```

Please note that only a pointer to the `setAlert` function is added. You don't execute the function in the dependencies array (that would add the return value of the function as a dependency, which is typically not the goal).



test

```
                react-dom.development.js:29742
Download the React DevTools for a better
development experience: https://reactjs.or
g/link/react-devtools
Expired!                          Alert.js:12
t                                 Alert.js:12
te                                Alert.js:12
tes                               Alert.js:12
test                              Alert.js:12
```

Figure 8.6: Multiple timers are started

Now, a new timer is started for every keystroke, and as a result, the entered message is output in the console.

Of course, this might also not be the desired result. You might only be interested in the final error message that was entered. This can be achieved by adding a cleanup function to the effect (and adjusting `setAlert` a little bit):

```
function setAlert() {
  return setTimeout(function () {
    console.log(alertMsg);
  }, 2000);
}

useEffect(
  function () {
    const timer = setAlert();

    return function () {
      clearTimeout(timer);
    };
  },
  [setAlert]
);
```

As shown in the *Cleaning Up after Effects* section, the timer is cleared with the help of a timer reference and `clearTimeout()` in the effect's cleanup function.

After adjusting the code like this, only the final alert message that was entered will be output.

Seeing the cleanup function in action again is helpful; the main takeaway is the importance of adding all dependencies, though—including function dependencies.

An alternative to including the function as a dependency would be to move the entire function definition into the effect function, because any value that's defined and used inside of an effect function must not be added as a dependency:

```
useEffect(
  function () {
    function setAlert() {
      return setTimeout(function () {
        console.log(alertMsg);
      }, 2000);
    }

    const timer = setAlert();

    return function () {
      clearTimeout(timer);
    };
  },
  []
);
```

Of course, you could also get rid of the `setAlert` function altogether then and just move the function's code into the effect function.

Either way, you will have to add a new dependency, `alertMsg`, which is now used inside of the effect function. Even though the `setAlert` function isn't a dependency anymore, you still must add any values used (and `alertMsg` is used in the effect function now):

```
useEffect(
  function () {
    function setAlert() {
      return setTimeout(function () {
        console.log(alertMsg);
      }, 2000);
    }

    const timer = setAlert();

    return function () {
      clearTimeout(timer);
    };
  },
  [alertMsg]
);
```

Hence, this alternative way of writing the code just comes down to personal preferences. It does not reduce the number of dependencies.

You would get rid of a function dependency if you were to move the function out of the component function. This is because, as mentioned in the *Unnecessary Dependencies* section, external dependencies (for example, those built into the browser or defined outside of component functions) should not be added as dependencies.

However, in the case of the `setAlert` function, this is not possible because `setAlert` uses `alertMsg`. Since `alertMsg` is a component state value, the function that uses it must be defined inside the component function; otherwise, it won't have access to that state value.

This can all sound quite confusing, but it comes down to two simple rules:

- Always add all non-external dependencies—no matter whether they're variables or functions.
- Functions are just objects and can change if their surrounding code executes again.

Avoiding Unnecessary Effect Executions

Since all dependencies should be added to `useEffect()`, you sometimes end up with code that causes an effect to execute unnecessarily.

Consider the example component below:

Note

The complete example can be found on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/08-effects/examples/unnecessary-executions`.

```
import { useState, useEffect } from 'react';

function Alert() {
  const [enteredEmail, setEnteredEmail] = useState('');
  const [enteredPassword, setEnteredPassword] = useState('');

  function handleUpdateEmail(event) {
    setEnteredEmail(event.target.value);
  }

  function handleUpdatePassword(event) {
    setEnteredPassword(event.target.value);
  }

  function validateEmail() {
    if (!enteredEmail.includes('@')) {
      console.log('Invalid email!');
    }
  }

  useEffect(function () {
    validateEmail();
  }, [validateEmail]);

  return (
    <form>
      <div>
        <label>Email</label>
        <input type="email" onChange={handleUpdateEmail} />
      </div>
      <div>
        <label>Password</label>
        <input type="password" onChange={handleUpdatePassword} />
      </div>
      <button>Save</button>
    </form>
  );
}

export default Alert;
```

This component contains a form with two inputs. The entered values are stored in two different state values (`enteredEmail` and `enteredPassword`). The `validateEmail()` function then performs some email validation and, if the email address is invalid, logs a message to the console. `validateEmail()` is executed with the help of `useEffect()`.

The problem with this code is that the effect function will be executed whenever `validateEmail` changes because, correctly, `validateEmail` was added as a dependency. But `validateEmail` will change whenever the component function is executed again. And that's not just the case for state changes to `enteredEmail` but also whenever `enteredPassword` changes—even though that state value is not used at all inside of `validateEmail`.

This unnecessary effect execution can be avoided with various solutions:

- You could move the code inside of `validateEmail` directly into the effect function (`enteredEmail` would then be the only dependency of the effect, avoiding effect executions when any other state changes).
- You could avoid using `useEffect()` altogether since you could perform email validation inside of `handleUpdateEmail`. Having `console.log()` (a side effect) in there would be acceptable since it wouldn't cause any harm.
- You could call `validateEmail()` directly in the component function—since it doesn't change any state, it wouldn't trigger an infinite loop.

Note

There is an article in the official React documentation that highlights scenarios where you might not need `useEffect()`: `https://react.dev/learn/you-might-not-need-an-effect`.

In addition, I created a video that summarizes the most important situations in which you do or do not need `useEffect()`: `https://www.youtube.com/watch?v=V1f8MOQiHRw`.

Of course, in some other scenarios, you might need to use `useEffect()`. Fortunately, React also offers a solution for situations like this: you can wrap the function that's used as a dependency with another React Hook, the `useCallback()` Hook.

The adjusted code would look like this:

```
import { useState, useEffect, useCallback } from 'react';

function Alert() {
  const [enteredEmail, setEnteredEmail] = useState('');
  const [enteredPassword, setEnteredPassword] = useState('');

  function handleUpdateEmail(event) {
    setEnteredEmail(event.target.value);
  }
  function handleUpdatePassword(event) {
    setEnteredPassword(event.target.value);
  }

  const validateEmail = useCallback(
    function () {
      if (!enteredEmail.includes('@')) {
        console.log('Invalid email!');
      }
    },
    [enteredEmail]
  );

  useEffect(
    function() {
      validateEmail();
    },
    [validateEmail]
  );

  // return JSX code ...
}

export default Alert;
```

`useCallback()`, like all React Hooks, is a function that's executed directly inside the component function. Like `useEffect()`, it accepts two arguments: another function (which can be anonymous or a named function) and a dependencies array.

Unlike `useEffect()`, though, `useCallback()` does not execute the received function. Instead, `useCallback()` ensures that a function is only recreated if one of the specified dependencies has changed. The default JavaScript behavior of creating a new function object whenever the surrounding code executes again is (synthetically) disabled.

`useCallback()` returns the latest saved function object. Hence, that returned value (which is a function) is saved in a variable or constant (`validateEmail` in the previous example).

Since the function wrapped by `useCallback()` now only changes when one of the dependencies changes, the returned function can be used as a dependency for `useEffect()` without executing that effect for all kinds of state changes or component updates.

In the case of the preceding example, the effect function would then only execute when `enteredEmail` changes—because that's the only change that will lead to a new `validateEmail` function object being created.

Another common reason for unnecessary effect execution is the usage of objects as dependencies, like in this example:

```
import { useEffect } from 'react';

function Error(props) {
  useEffect(
    function () {
      // performing some error logging
      // in a real app, a HTTP request might be sent to some analytics API
      console.log('An error occurred!');
      console.log(props.message);
    },
    [props]
  );

  return <p>{props.message}</p>;
}

export default Error;
```

This `Error` component is used in another component, the `Form` component, like this:

```
import { useState } from 'react';

import Error from './Error.jsx';

function Form() {
  const [enteredEmail, setEnteredEmail] = useState('');
  const [errorMessage, setErrorMessage] = useState('');

  function handleUpdateEmail(event) {
    setEnteredEmail(event.target.value);
  }

  function handleSubmitForm(event) {
    event.preventDefault();
    if (!enteredEmail.endsWith('.com')) {
      setErrorMessage('Only email addresses ending with .com are accepted!');
    }
  }

  return (
    <form onSubmit={handleSubmitForm}>
      <div>
        <label>Email</label>
        <input type="email" onChange={handleUpdateEmail} />
      </div>
      {errorMessage && <Error message={errorMessage} />}
      <button>Submit</button>
    </form>
  );
}

export default Form;
```

The `Error` component receives an error message via props (`props.message`) and displays it on the screen. In addition, with the help of `useEffect()`, it does some error logging. In this example, the error is simply output to the JavaScript console. In a real app, the error might be sent to some analytics API via an HTTP request. Either way, a side effect that depends on the error message is performed.

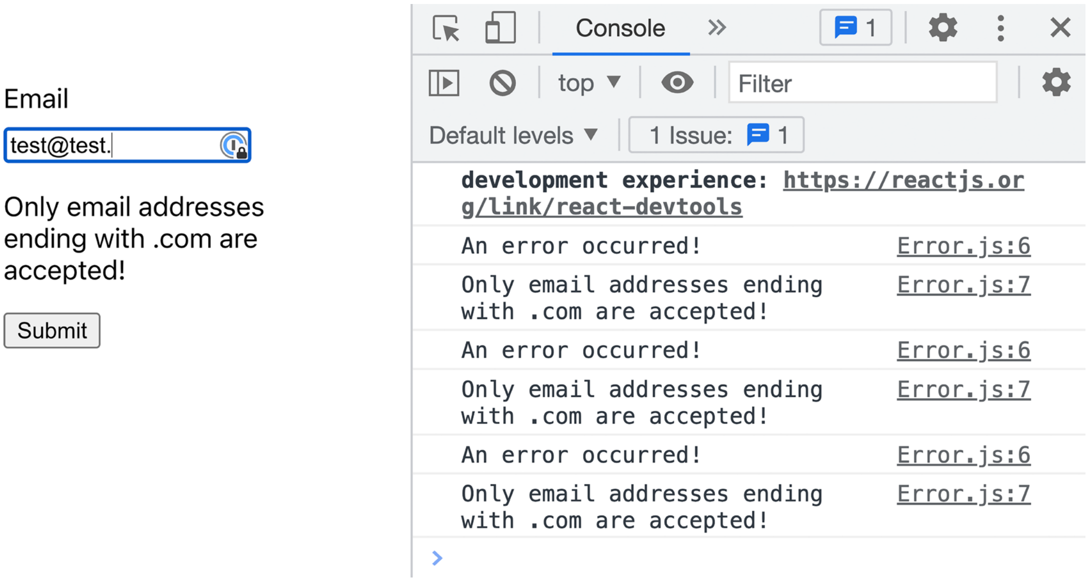
The `Form` component contains two state values, tracking the entered email address as well as the error status of the input. If an invalid input value is submitted, `errorMessage` is set and the `Error` component is displayed.

The interesting part about this example is the dependency array of `useEffect()` inside the `Error` component. It contains the `props` object as a dependency (`props` is always an object, grouping all prop values together). When using objects (props or any other object; it does not matter) as dependencies for `useEffect()`, unnecessary effect function executions can be the result.

You can see this problem in this example. If you run the app and enter an invalid email address (e.g., `test@test.de`), you'll notice that subsequent keystrokes in the email input field will cause the error message to be logged (via the effect function) for every keystroke.

Note

The full code can be found on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/08-effects/examples/objects-as-dependencies`.



Email

test@test.

Only email addresses ending with .com are accepted!

```
development experience: https://reactjs.org/link/react-devtools
An error occurred!                          Error.js:6
Only email addresses ending                 Error.js:7
with .com are accepted!
An error occurred!                          Error.js:6
Only email addresses ending                 Error.js:7
with .com are accepted!
An error occurred!                          Error.js:6
Only email addresses ending                 Error.js:7
with .com are accepted!
```

Figure 8.7: A new error message is logged for every keystroke

Those extra executions can occur because component re-evaluations (i.e., component functions being invoked again by React) will produce brand-new JavaScript objects. Even if the values of properties of those objects did not change (as in the preceding example), technically, a brand-new object in memory is created by JavaScript. Since the effect depends on the entire object, React only "sees" that there is a new version of that object and hence runs the effect again.

In the preceding example, a new props object (for the `Error` component) is created whenever the `Form` component function is called by React—even if the error message (the only prop value that's set) did not change.

In this example, that's just annoying since it clutters the JavaScript console in the developer tools. However, if you were sending an HTTP request to some analytics backend API, it could cause bandwidth problems and make the app slower. Therefore, it's best if you get into the habit of avoiding unnecessary effect executions as a general rule.

In the case of object dependencies, the best way to avoid unnecessary executions is to simply destructure the object so that you can pass only those object properties as dependencies that are needed by the effect:

```
function Error(props) {
  const { message } = props; // destructure to extract required properties

  useEffect(
    function () {
      console.log('An error occurred!');
      console.log(message);
    },
    // [props] // don't use the entire props object!
    [message]
  );

  return <p>{message}</p>;
}
```

In the case of props, you could also destructure the object right in the component function parameter list:

```
function Error({message}) {
  // ...
}
```

Using this approach, you ensure that only the required property values are set as dependencies. Therefore, even if the object gets recreated, the property value (in this case, the value of the `message` property) is the only thing that matters. If it doesn't change, the effect function won't be executed again.

Effects and Asynchronous Code

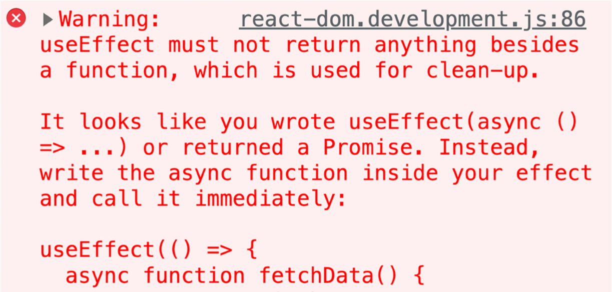
Some effects deal with asynchronous code (sending HTTP requests is a typical example). When performing asynchronous tasks in effect functions, there is one important rule to keep in mind, though: the effect function itself should not be asynchronous and should not return a promise. This does not mean that you can't work with promises in effects—you just must not return a promise.

You might want to use `async`/`await` to simplify asynchronous code, but when doing so inside of an effect function, it's easy to accidentally return a promise. For example, the following code would work but does not follow best practices:

```
useEffect(async function () {
  const fetchedPosts = await fetchPosts();
  setLoadedPosts(fetchedPosts);
}, []);
```

Adding the `async` keyword in front of `function` unlocks the usage of `await` inside the function—which makes dealing with asynchronous code (that is, with promises) more convenient.

But the effect function passed to `useEffect()` should only return a normal function, if anything. It should not return a promise. Indeed, React actually issues a warning when trying to run code like the preceding snippet:



```
Warning:      react-dom.development.js:86
useEffect must not return anything besides
a function, which is used for clean-up.

It looks like you wrote useEffect(async ()
=> ...) or returned a Promise. Instead,
write the async function inside your effect
and call it immediately:

useEffect(() => {
  async function fetchData() {
```

Figure 8.8: React shows a warning about async being used in an effect function

To avoid this warning, you can use promises without `async`/`await`, like this:

```
useEffect(function () {
  fetchPosts().then((fetchedPosts) => setLoadedPosts(fetchedPosts));
}, []);
```

This works because the effect function doesn't return the promise.

Alternatively, if you want to use `async`/`await`, you can create a separate wrapper function inside of the effect function, which is then executed in the effect:

```
useEffect(function () {
  async function loadData() {
    const fetchedPosts = await fetchPosts();
    setLoadedPosts(fetchedPosts);
  }

  loadData();
}, []);
```

By doing that, the effect function itself is not asynchronous (it does not return a promise), but you can still use `async`/`await`.

Rules of Hooks

In this chapter, two new Hooks were introduced: `useEffect()` and `useCallback()`. Both Hooks are very important—`useEffect()` especially, as this is a Hook you will typically use a lot. Together with `useState()` (introduced in *Chapter 4*, *Working with Events and State*) and `useRef()` (introduced in *Chapter 7*, *Portals and Refs*), you now have a solid set of key React Hooks.

When working with React Hooks, there are two rules (the so-called **rules of Hooks**) you must follow:

- Only call Hooks at the top level of component functions. Don't call them inside of `if` statements, loops, or nested functions.
- Only call Hooks inside of React components or custom Hooks (custom Hooks will be covered in *Chapter 12*, *Building Custom React Hooks*).

These rules exist because React Hooks won't work as intended if used in a non-compliant way. Fortunately, React will generate a warning message if you violate one of these rules; hence, you will notice if you accidentally do so.

Summary and Key Takeaways

- Actions that are not directly related to the main process of a function can be considered side effects.
- Side effects can be asynchronous tasks (for example, sending an HTTP request), but can also be synchronous (for example, `console.log()` or accessing browser storage).
- Side effects are often needed to achieve a certain goal, but it's a good idea to separate them from the main process of a function.
- Side effects can become problematic if they cause infinite loops (because of the update cycles between effect and state).
- `useEffect()` is a React Hook that should be used to wrap side effects and perform them in a safe way.
- `useEffect()` takes an effect function and an array of effect dependencies.
- The effect function is executed directly after the component function is invoked (not simultaneously).
- Any value, variable, or function used inside of an effect should be added to the dependencies array.
- Dependency array exceptions are external values (defined outside of a component function), state updating functions, or values defined and used inside of the effect function.
- If no dependency array is specified, the effect function executes after every component function invocation.
- If an empty dependency array is specified, the effect function runs once when the component first mounts (that is, when it is created for the first time).
- Effect functions can also return optional cleanup functions that are called right before an effect function is executed again (and right before a component is removed from the DOM).
- Effect functions must not return promises.
- For function dependencies, `useCallback()` can help reduce the number of effect executions.
- For object dependencies, destructuring can help reduce the number of effect executions.

What's Next?

Dealing with side effects is a common problem when building apps because most apps need some kind of side effects (for example, sending an HTTP request) to work correctly. Therefore, side effects aren't a problem themselves, but they can cause problems (for example, infinite loops) if handled incorrectly.

With the knowledge gained in this chapter, you know how to handle side effects efficiently with `useEffect()` and related key concepts.

Many side effects are triggered because of user input or interaction—for example, because some form was submitted. The next chapter will revisit the concept of form submissions by exploring React's **form actions** feature.

Test Your Knowledge!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to examples that can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/08-effects/exercises/questions-answers.md`:

1. How would you define a side effect?
2. What's a potential problem that could arise with some side effects in React components?
3. How does the `useEffect()` Hook work?
4. Which values should *not* be added to the `useEffect()` dependencies array?
5. Which value can be returned by the effect function? And which kind of value *must not* be returned?

Apply What You Learned

Now that you know about effects, you can add even more exciting features to your React apps. Fetching data via HTTP upon rendering a component is just as easy as accessing browser storage when some state changes.

In the following section, you'll find an activity that allows you to practice working with effects and `useEffect()`. As always, you will need to employ some of the concepts covered in earlier chapters (such as working with state).

Activity 8.1: Building a Basic Blog

In this activity, you must add logic to an existing React app to render a list of blog post titles fetched from a backend web API and submit newly added blog posts to that same API. The backend API used is `https://jsonplaceholder.typicode.com/`, which is a dummy API that doesn't actually store any data you send to it. It will always return the same dummy data, but it's perfect for practicing sending HTTP requests.

As a bonus, you can also add logic to change the text of the submit button while the HTTP request to save the new blog post is on its way.

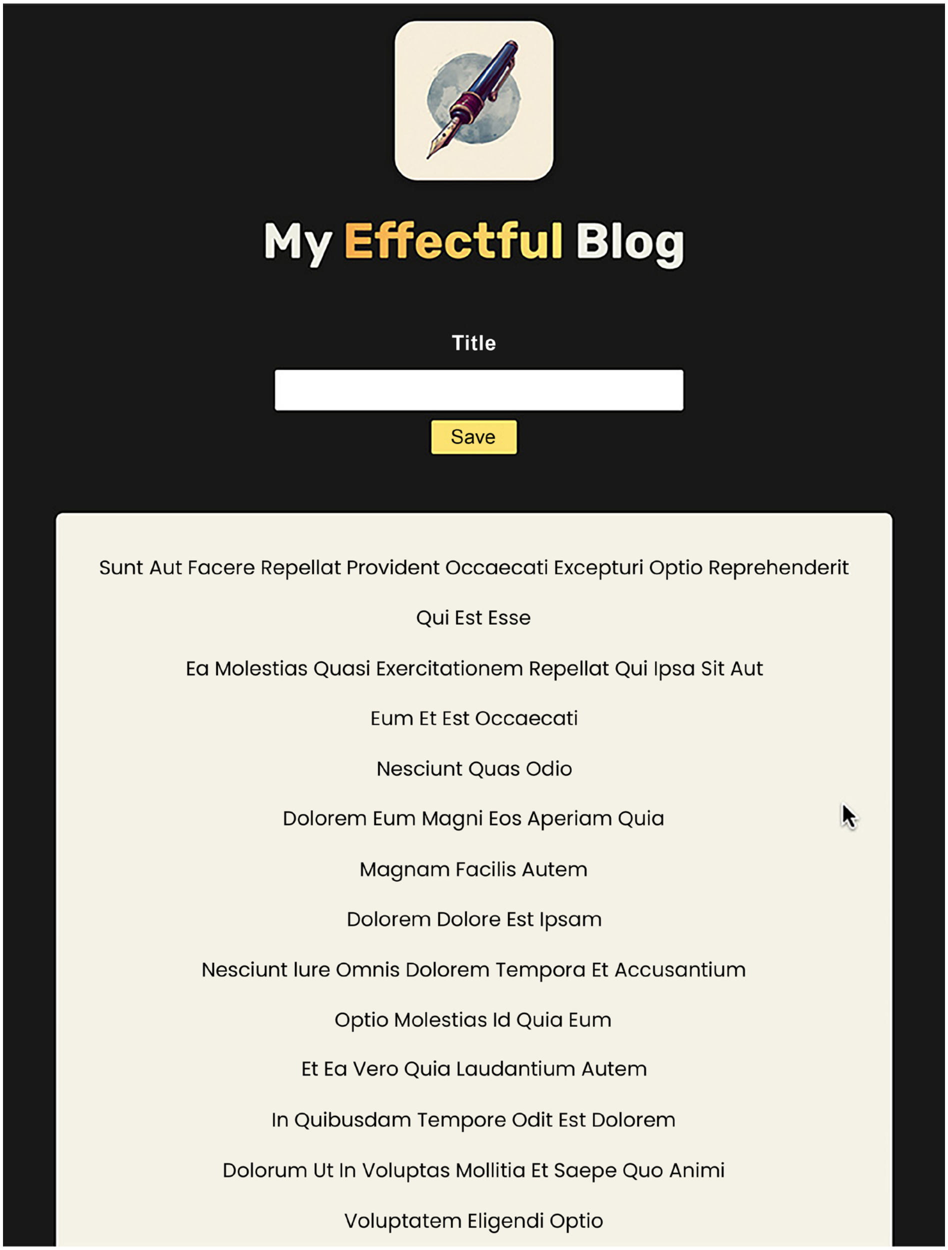
Use your knowledge about effects and browser-side HTTP requests to implement a solution.

> **Note**
>
> You can find the starting code for this activity at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/08-effects/activities/practice-1-start`. When downloading this code, you'll always download the entire repository. Make sure to then navigate to the subfolder with the starting code (`activities/practice-1-start`, in this case) to use the right code snapshot.
>
> For this activity, you need to know how to send HTTP requests (`GET`, `POST`, and so on) via JavaScript (for example, via the `fetch()` function or with the help of a third-party library). If you don't have that knowledge yet, this resource can get you started: `http://packt.link/DJ6Hx`.

After downloading the code and running `npm install` in the project folder to install all required dependencies, the solution steps are as follows:

1. Send a `GET` HTTP request to the dummy API to fetch blog posts inside the `App` component (when the component is first rendered).
2. Display the fetched dummy blog posts on the screen.
3. Handle form submissions and send a `POST` HTTP request (with some dummy data) to the dummy backend API.
4. Bonus: Set the button caption to `Saving…` while the request is on its way (and to `Save` when it's not).

The expected result should be a user interface that looks like this:



My Effectful Blog

Sunt Aut Facere Repellat Provident Occaecati Excepturi Optio Reprehenderit

Qui Est Esse

Ea Molestias Quasi Exercitationem Repellat Qui Ipsa Sit Aut

Eum Et Est Occaecati

Nesciunt Quas Odio

Dolorem Eum Magni Eos Aperiam Quia

Magnam Facilis Autem

Dolorem Dolore Est Ipsam

Nesciunt Iure Omnis Dolorem Tempora Et Accusantium

Optio Molestias Id Quia Eum

Et Ea Vero Quia Laudantium Autem

In Quibusdam Tempore Odit Est Dolorem

Dolorum Ut In Voluptas Mollitia Et Saepe Quo Animi

Voluptatem Eligendi Optio

Figure 8.9: The final user interface

Note

You will find a full example solution here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/08-effects/activities/practice-1`.

9

Handling User Input & Forms with Form Actions

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Describe the purpose of React form actions
- Build and use custom form actions to handle form submissions
- Use the `useActionState()` Hook to manage form-dependent state
- Render a pending UI during submission via the `useFormStatus()` Hook
- Perform optimistic state updates with the `useOptimistic()` Hook
- Implement both synchronous and asynchronous actions

Introduction

In *Chapter 4, Working with Events and State*, you learned how to handle form submissions in React applications. And while there is absolutely nothing wrong with the approach shown there—indeed, it's arguably the approach you'll find in the majority of React projects—React provides an alternative way of handling form submissions when working in projects that use React version 19 or later. React 19 introduced a new feature called **actions** (also referred to as **form actions** throughout this chapter) that can simplify the process of handling form submissions, extracting user input, and providing validation feedback.

This chapter will first revisit form submissions as introduced in *Chapter 4* and explore how user input can be extracted and validated. Thereafter, this chapter will introduce form actions and explain how to perform the same steps (handle submission, extract values, and validate values) using that feature. You will also learn about action-related React Hooks like `useActionState()`.

Handling Form Submissions without Actions

As you learned in *Chapter 4*, *Working with Events and State*, when not using actions, you can handle form submissions by listening to the `submit` event via the `onSubmit` prop on the `<form>` element.

Consider the following example code snippet:

```
function App() {
  function handleSubmit(event) {
    event.preventDefault();
    console.log('Submitted!');
  }

  return (
    <form onSubmit={handleSubmit}>
      <p>
        <label htmlFor="email">Email</label>
        <input type="email" id="email" />
      </p>
      <p>
        <label htmlFor="password">Password</label>
        <input type="password" id="password" />
      </p>
      <p className="actions">
        <button>Login</button>
      </p>
    </form>
  );
}
```

You find the full working example on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/09-form-actions/examples/01-form-submission-without-actions`.

This code displays a form and handles its submission via the `handleSubmit()` function. This function automatically receives an `event` object, which is used to prevent the browser's default behavior of sending an HTTP request to the server hosting the website.

But, of course, just handling the submission isn't too useful. Typically, you also want to extract and use values entered by the website user.

Extracting User Input

When it comes to extracting values entered into a form, you have a couple of options:

- Track the values via state (i.e., by using `useState()`), as described in *Chapter 4*.

- Rely on Refs via `useRef()`, as explained in *Chapter 7*, *Portals and Refs*.
- Take advantage of the automatically created `event` object.

Tracking State

You can track the values entered by the user via state managed by `useState()`, as explained in *Chapter 4*. For example, the form input values from the previous code snippet can be tracked and used in `handleSubmit()`, as shown in the following example:

```
function App() {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');

  function handleSubmit(event) {
    event.preventDefault();
    const credentials = { email, password };
    console.log(credentials);
  }

  function handleEmailChange(event) {
    setEmail(event.target.value);
  }

  function handlePasswordChange(event) {
    setPassword(event.target.value);
  }

  return (
    <form onSubmit={handleSubmit}>
      <p>
        <label htmlFor="email">Email</label>
        <input
          type="email"
          id="email"
          value={email}
          onChange={handleEmailChange}
        />
      </p>
      <p>
        <label htmlFor="password">Password</label>
        <input
          type="password"
          id="password"
          value={password}
          onChange={handlePasswordChange}
        />
      </p>
      <p className="actions">
        <button>Login</button>
      </p>
    </form>
  );
}
```

In this updated example code snippet, the `useState()` Hook is used to manage `email` and `password` state values. The state values are updated with every keystroke on the input fields. As a result, the latest entered values are available inside of `handleSubmit()` when the form is submitted.

This approach works well and will be found in many React projects. However, there are some potential downsides to using state to track input values:

- Since the state is updated on every keystroke, and the component function is re-executed whenever some state value changes, application performance could suffer.
- When working with more complex forms with more input fields, a lot of different state values may need to be managed.

You can work around these issues by implementing code optimizations, which will be discussed in *Chapter 10*, *Behind the Scenes of React and Optimization Opportunities*, and by managing state as an object, as explained in *Chapter 11*, *Working with Complex State*.

But you could also consider using Refs to extract input values.

Relying on Refs

If you're building a form where you don't plan on setting input values, and where you instead only want to read those values upon form submission, using React's ref feature (introduced in *Chapter 7*) might make sense:

```
function App() {
  const emailRef = useRef(null);
  const passwordRef = useRef(null);

  function handleSubmit(event) {
    event.preventDefault();
    const credentials = {
      email: emailRef.current.value,
      password: passwordRef.current.value,
    };
    console.log(credentials);
    }

    return (
      <form onSubmit={handleSubmit}>
        <p>
          <label htmlFor="email">Email</label>
          <input type="email" id="email" ref={emailRef} />
        </p>
        <p>
          <label htmlFor="password">Password</label>
          <input type="password" id="password" ref={passwordRef} />
        </p>
        <p className="actions">
          <button>Login</button>
        </p>
      </form>
    );
  }
```

In this code block, the `useRef()` Hook is used to create two Refs that are connected to the email and password input fields. These Refs are then used to read the entered values inside of `handleSubmit()`.

When using this approach, the `App` component function is not executed with every keystroke anymore. But you still have to write the code where the Refs are created via `useRef()` and where they are connected to the JSX elements via the `ref` prop.

That's why you could consider relying on the browser and the automatically created `event` object (which is received in `handleSubmit()`), instead of using React features to extract those entered values.

Taking Advantage of the event Object

In *Chapter 4*, *Working with Events and State*, you learned that the browser tries to send an HTTP request when a form is submitted. That's why `event.preventDefault()` is called inside of `handleSubmit()`—this function call ensures that this request is not sent.

However, the `event` object is not just useful for preventing that default. It also carries important information about the `submit` event that occurred. For example, you can get access to the underlying form DOM object (i.e., a JavaScript object that describes the rendered `<form>` element, its configuration, and its current status) via `event.currentTarget`.

This is very useful because you can pass that form DOM object to the `FormData` constructor function that is provided by the browser. This interface can be used to extract a form's input field values.

The following example shows the concrete usage of this feature:

```
function App() {
  function handleSubmit(event) {
      event.preventDefault();
      const fd = new FormData(event.currentTarget);
      const credentials = {
        email: fd.get('email'),
        password: fd.get('password'),
      };
      console.log(credentials);
    }

    return (
      <form onSubmit={handleSubmit}>
        <p>
          <label htmlFor="email">Email</label>
          <input type="email" id="email" name="email" />
        </p>
        <p>
          <label htmlFor="password">Password</label>
          <input type="password" id="password" name="password" />
        </p>
        <p className="actions">
          <button>Login</button>
        </p>
      </form>
    );
  }
```

As you can see in the above code snippet, the form data object `fd` is constructed by instantiating `FormData`. As mentioned, the `FormData` interface is provided by the browser; hence, it doesn't need to be imported from React or any other library.

This form data object offers various methods that help with accessing form field values—for example, the `get()` method to extract the value of a specific input field. In order to identify the input field for which you want to get hold of the value, the `get()` method requires the name of the input field as an argument. That's why you must also set the `name` prop on the form control elements (i.e., on the `<input>` elements in the above example).

This approach has the advantage that you need neither state nor Refs; hence, slightly less code must be written. In addition, since almost no React features are used, this code will be less prone to break due to possible future React changes.

Consequently, this approach might look like the best way of handling form submissions. But is it?

Which Solution Is Best?

There is no right or wrong way of handling form submissions. Besides personal preference, application requirements also might favor one approach over the others.

For example, if your application needs to change the input values, using only `FormData` as shown above would not be ideal, since you would have to write imperative code to update an input field.

That's a problem because, as explained in *Chapter 1*, *React – What and Why?*, you should avoid writing code like this in your React apps:

```
function clearInput() {
  document.getElementById('email').value = ''; // imperative code :(
}
```

Thus, if you need to edit an input's value, using state (i.e., `useState()`) is preferable:

```
const [email, setEmail] = useState('');
// ... other code
function clearInput() {
  setEmail('');
}

// simplified JSX code below
return (
  <form>
    <input
      value={email}
      onChange={event => setEmail(event.target.value)} />
  </form>
);
```

Even if you don't need to update any input fields, the `event` object and `FormData` alone might not do the trick.

For example, if you need to access the input fields outside of `handleSubmit()`, the `event` object is not available. As a result, interacting with the form element and its child elements is not possible via the `event` object. In such scenarios, working with Refs that are directly connected to the individual input elements will likely simplify things.

The following example uses a ref to call the `<input>` element's built-in `focus()` method inside of a function:

```
const emailRef = useRef(null);

function showForm() {
  // other code ...
```

```
    emailRef.current.focus();
  }

  // simplified JSX code below
  return (
    <form>
      <input ref={emailRef} />
    </form>
  );
```

So, as you can see, there is no silver bullet. All these React features and different ways of handling form submissions exist for good reasons. You can mix and match them as needed; therefore, it's helpful to be aware of these different options.

But even though there are already a couple of ways of handling form submissions, with React 19, there's yet another one.

Handling Form Submissions with Actions

React 19 introduced the concept of (form) actions—a concept that actually consists of two kinds of actions: **client actions** and **server actions**. Both types of actions can help with handling form submissions, but for the purpose of this chapter, the term **form actions** will be used to describe client actions (i.e., form actions that execute in the website user's browser). Server actions will be covered separately in *Chapter 16*, *React Server Components & Server Actions*.

Form actions were introduced to simplify the process of handling form submissions and data extraction—especially when building full stack apps with server actions. Furthermore, they can also be very useful when combined with some new React Hooks, which will be discussed later in this chapter.

Here's how a form submission can be handled via a client form action:

```
function App() {
  function submitAction(formData) {
    const credentials = {
      email: formData.get('email'),
      password: formData.get('password'),
    };
    console.log(credentials);
  }

  return (
    <form action={submitAction}>
      <p>
        <label htmlFor="email">Email</label>
```

```
          <input type="email" id="email" name="email" />
        </p>
        <p>
          <label htmlFor="password">Password</label>
          <input type="password" id="password" name="password" />
        </p>
        <p className="actions">
          <button>Login</button>
        </p>
      </form>
    );
  }
```

At first sight, this example may look very similar to the code snippet where the `event` object and `currentTarget` were used to derive the `FormData`. But if you take a closer look, you'll see that there are some key differences:

- `handleSubmit` was renamed `submitAction` and accepts a parameter named `formData` instead of `event`.
- The `<form>` element no longer has the `onSubmit` prop—instead, it now has an `action` prop that points at the `submitAction` function.

The name change of the function is optional; there is no technical requirement to name this function `submitAction` or anything like that. But changing the name makes sense because the function no longer directly handles the `submit` event. Instead, it's used as a value for the newly added `action` prop.

And that's precisely what React's form action feature is all about: setting the `action` prop of a `<form>` element to a function that React will then invoke on your behalf when the form is submitted. However, unlike when using the `onSubmit` prop, React will prevent the browser default and create a form data object for you (and pass that object as an argument to the action function).

You no longer have to perform these steps manually, and as a result, the form submission can be handled with a minimal amount of code.

Of course, if you need to set and manage the input values manually, or if you need to interact with the form fields at some point (e.g., to call `focus()`), you'll still need to work with state or Refs. But if you're just trying to handle the submission and get hold of the entered values, using the form actions feature can be very handy.

But form actions are not just useful because they may require less code.

Synchronous vs Asynchronous Actions

Client form actions can be either synchronous or asynchronous, which means you can also use and return a `Promise` in the action function. Therefore, you can also use `async / await` with that function.

For example, if you have a form in an application that aims to store some task data in the browser's storage (via the `localStorage` API), you can do that with a synchronous action (since `localStorage` is a synchronous API):

```
function storeTaskAction(formData) {
  const task = {
    title: formData.get('title'),
    body: formData.get('body'),
    dueDate: formData.get('date')
  };
  localStorage.setItem('daily-task', JSON.stringify(task));
}
```

This action function is synchronous, since it doesn't return a `Promise` or use `async / await`. Therefore, as you can see, all form action examples thus far have used synchronous actions.

But if you're working on a project that needs to submit entered data to a backend via an HTTP request, you can take advantage of the support for asynchronous code:

```
async function storeTodoAction(formData) {
  const todoTitle = formData.get('title');
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/todos',
    {
      method: 'POST',
      body: JSON.stringify({ title: todoTitle }),
      headers: {
        'Content-type': 'application/json; charset=UTF-8',
      },
    }
  );
  const todo = await response.json();
  console.log(todo);
}
```

In this example, the `async` keyword is added in front of the function. This converts the function into an asynchronous one that will return a `Promise`.

This flexibility offered by React's form actions feature is very useful, since it allows you to perform a broad variety of operations upon form submission. However, it is important to keep in mind that, for now, all these actions always execute on the client side, i.e., in the browser of the website visitor. Server-side actions will be explored in *Chapter 16*.

Behind the Scenes: Actions Are Transitions

Before diving deeper into form actions, it may be helpful to take a brief look under the hood.

This is because, technically, actions (i.e., both client and server actions) in React are so-called **transitions**. To be precise, they are asynchronous transitions.

Thus, the question is, what's a transition in React?

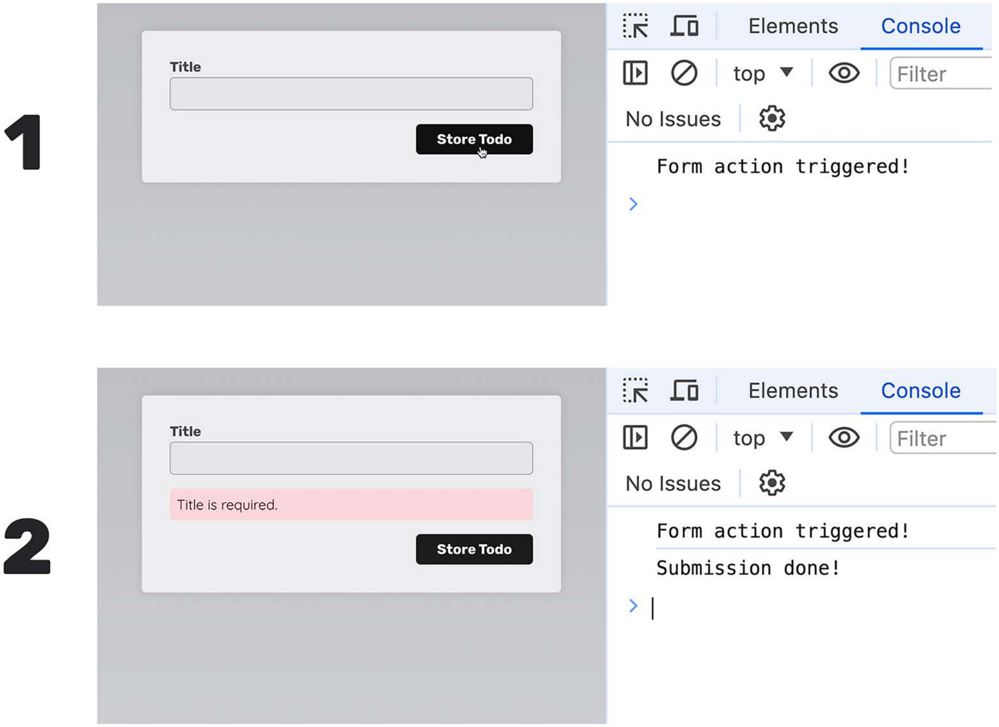
In a React app, a transition is a concept, where React will ensure that some potentially time-consuming state updates will not block UI updates.

Form actions can be considered (potentially) time-consuming state updates; hence, under the hood, React handles them such that the remaining UI will stay responsive.

As a result, any state updating calls you make inside a form action function will only be processed by React once that form action is done. For example, the following code will, probably unexpectedly, only update the UI after three seconds:

```
import { useState } from 'react';

function App() {
  const [error, setError] = useState(null);

  async function storeTodoAction(formData) {
    const todoTitle = formData.get('title');
    if (!todoTitle || todoTitle.trim() === '') {
      setError('Title is required.'); // state update BEFORE delay
    }
    // 3s delay to simulate a slow process
    await new Promise((resolve) => setTimeout(resolve, 3000));
    console.log('Submission done!');
  }

  return (
    <>
      <form action={storeTodoAction}>
        <p>
          <label htmlFor="title">Title</label>
          <input type="text" id="title" name="title" />
        </p>
        {error && <p className="errors">{error}</p>}
        <p className="actions">
          <button>Store Todo</button>
        </p>
      </form>
    </>
  );
}
```

Even though the `error` state is updated before the delay starts, React will not re-execute the component function (and, therefore, update the UI) before the form action as a whole is done. Therefore, the error message only shows up on the screen after three seconds.


1
Title
Store Todo
Elements
Console
top
Filter
No Issues
2
Title
Title is required.
Store Todo
Elements
Console
top
Filter
No Issues

```
Form action triggered!
```

```
Form action triggered!
Submission done!
```

Figure 9.1: The error message only shows up with a delay

Note

You find the complete example code on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/09-form-actions/examples/08-transition`.

Managing State Based on Form Submissions

When handling form submissions, it's quite common that you may also want to update the UI after the submission. For asynchronous actions, where the executed operation may take a couple of seconds (depending on the operation, of course), you might even want to update the UI during the submission, showing some pending state while the submitted form is being processed.

React aims to help you with both requirements by offering two specific form action-related Hooks: `useActionState()` and `useFormStatus()`.

Updating UI State with useActionState()

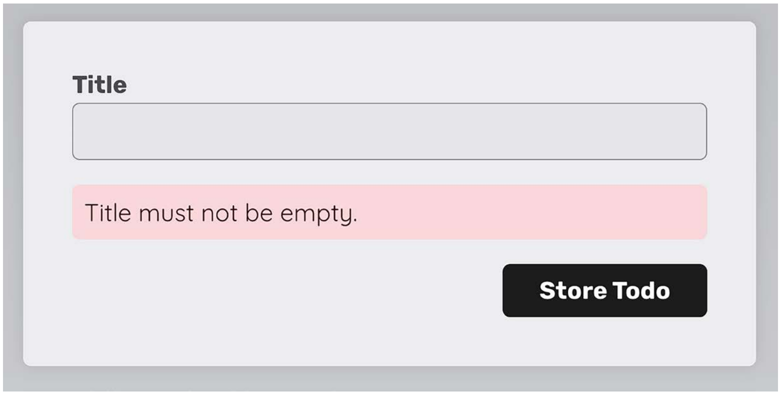
React provides a Hook called `useActionState()`, which is meant to be used in conjunction with form actions—no matter whether you work with client or server actions.

The goal of this Hook is to help you update the application's UI based on the result of a form action.

This can, for example, be helpful to validate form input values and show an error message if there is invalid input. To perform this task, the `useActionState()` Hook can be imported from the `react` package and used like this:

```
import { useActionState } from 'react';

function App() {
  async function storeTodoAction(prevState, formData) {
    const todoTitle = formData.get('title');

    if (!todoTitle || todoTitle.trim() === '') {
      return {
        error: 'Title must not be empty.',
      };
    }

    // sending HTTP request etc...
    return {
      error: null,
    };
  }

  const [formState, formAction] = useActionState(storeTodoAction, {
    error: null,
  });
  return (
    <form action={formAction}>
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" name="title" />
      </p>
      {formState.error && <p className='errors'>
        {formState.error}
      </p>}
      <p className="actions">
        <button>Store Todo</button>
      </p>
    </form>
  );
}
```

When running this example application, users will see validation error messages if there is invalid input.



Title must not be empty.

Figure 9.2: An error message is shown when submitting an empty input field

A couple of things are going on in this code example:

- The form action function was changed to accept two parameters instead of just one: a previous state (`prevState`) and the submitted data (`formData`).
- The form action now also returns a value: an object with a key named `error` that contains an error message or `null`.
- The `useActionState()` Hook is imported and used: it receives the form action function (`storeTodoAction`) as a first argument, and some initial state object (`{error: null}` in this case) as a second argument.
- The `useActionState()` Hook also returns a value: an array from which two elements are destructured (`formState` and `formAction`).

- The destructured `formAction` replaces `storeTodoAction` as a value for the `<form>`'s `action` prop.
- `formState` is used to conditionally display the value stored in the `error` key of `formState`.

So, as you can see, `useActionState()` is a Hook that expects a form action function (synchronous or asynchronous) as a first argument and an initial state as a second input. That initial state is required to have some state available if the form has not been submitted yet. After form submission, the initial state will be replaced by new state values returned by the form action function.

Since the purpose of `useActionState()` is to provide some state value that can be used to update (parts of) the UI, that derived state is exposed via the value returned by `useActionState()`:

```
const [formState, formAction] = useActionState(storeTodoAction, {
    error: null,
  }
);
```

That returned value is an array with exactly three elements, in the following order:

1. The current state value, which is either the initial state (if the form wasn't submitted yet) or the state value returned by the form action function.
2. An updated form action function, which is essentially your action function, wrapped by React. This is necessary so that React gets access to the value returned by your action function (which is the new state).
3. A boolean value that indicates whether the form is currently being submitted or not. This third element is not used in the previous code example and will be discussed in the *Managing Pending UI State* section of this chapter.

Therefore, when using `useActionState()`, you no longer bind your action function to the `action` prop of the `<form>` element. Instead, you use the action function created by `useActionState()`—i.e., you use the action function that wraps your action function.

When using `useActionState()`, you also must adjust your form action function because React will call your function with two arguments instead of just one: the previous state and the submitted form data:

```
async function storeTodoAction(prevState, formData) {
  // ...
}
```

The previous form state is passed to your action function so that you can use it to derive your new state from it (in conjunction with the submitted form data). In the above example, this is actually not the case—the previous state parameter is not used there. It must be accepted as a parameter nonetheless.

However, that's not the only change made to the form action function. Instead, it should now also return a new state value that will then be exposed to the component function by `useActionState()` (via the first element in the array returned by `useActionState()`):

```
async function storeTodoAction(prevState, formData) {
  // ...
  return {
    error: 'Title must not be empty.'
  };
}
```

That state value can be anything—a string, a number, an array, an object, etc. In the previous code example, it's an object with a key named `error` that holds either `null` or a string error message.

Whenever the form is submitted, and the form action function is therefore executed or returns a value, `useActionState()` will trigger React to re-execute the surrounding component function. Hence, the updated state is made available. If that sounds familiar to `useState()`, you're right! `useActionState()` is essentially like `useState()`, fine-tuned to derive state from actions.

`useActionState()` is, therefore, definitely an important Hook, although it's actually not limited to just exposing the values returned by your actions to component functions.

Managing Pending UI State with useActionState()

Consider a scenario where you have a form action that takes a couple of seconds to finish its operation. For example, you could have an action that sends a request to a slow server or via a slow internet connection. In such scenarios, you might want to update the UI during the form submission to show the user that something is happening.

In the following example, a function named `saveTodo()` is called from inside the form action. That function adds a deliberate delay of three seconds to simulate a slow network or server:

```
async function saveTodo(todo) {
  // dummy function that simulates a slow backend which manages todos
  await new Promise((resolve) => setTimeout(resolve, 3000)); // delay
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/todos', {
      method: 'POST',
      body: JSON.stringify(todo),
      headers: {
        'Content-type': 'application/json; charset=UTF-8',
      },
    }
  );
  const fetchedTodo = await response.json();
  console.log(fetchedTodo);
}

function App() {
  async function storeTodoAction(prevState, formData) {
    const todoTitle = formData.get('title');
    if (!todoTitle || todoTitle.trim() === '') {
      return {
        error: 'Title must not be empty.',
      };
    }

    await saveTodo({ title: todoTitle });
    return {
      error: null,
    };
  }

  // same code as before, hence omitted
}
```

When using form actions, like in this example, updating the UI while the form submission is handled is relatively easy because `useActionState()` exposes a third element in its returned array: a boolean that indicates whether the action is currently executing or not.

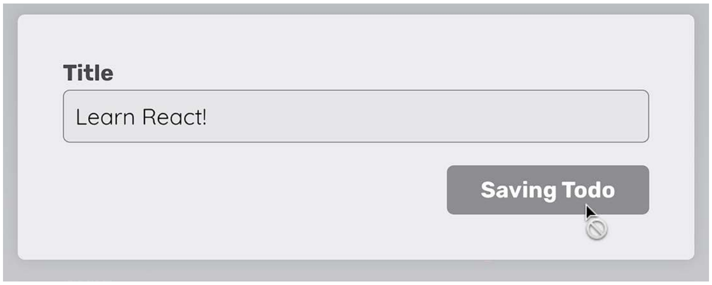
The above example can, therefore, be adjusted like this to take advantage of that boolean value:

```
function App() {
  async function storeTodoAction(prevState, formData) {
    // same code as before, hence omitted
  }

  const [formState, formAction, pending] = useActionState(
    storeTodoAction,
    {
      error: null,
    }
  );

  return (
    <form action={formAction}>
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" name="title" />
      </p>
      {formState.error &&
        <p className="errors">{formState.error}</p>}
      <p className="actions">
```

```
        <button disabled={pending}>
          {pending ? 'Saving' : 'Store'} Todo
        </button>
      </p>
    </form>
  );
}
```

The `pending` element is retrieved from the array via destructuring, and then it is used to disable the `<button>` and update the button text.

As a result, the UI changes once the form is submitted—until it's done after three seconds (in this case, due to the delay added in the `saveTodo()` function earlier).



Title

Learn React!

Figure 9.3: The button is disabled and shows Saving Todo fallback text during form submission

Handling Pending UI State with useFormStatus()

The `pending` element returned by `useActionState()` is a simple and straightforward, but not the only, way of updating the UI while a form action executes.

React also offers a `useFormStatus()` Hook that provides information about the current form submission status. To be precise, it's the `react-dom` package (not `react`!) that exports this `useFormStatus()` Hook.

Unlike `useActionState()`, `useFormStatus()` must be called in some nested component that's wrapped by the `<form>` element whose submission status you're interested in.

You could, for example, build a `SubmitButton` component that's defined and used as shown in this code snippet:

```
import { useFormStatus } from 'react-dom';

import { saveTodo } from './todos.js';

function SubmitButton() {
  const { pending } = useFormStatus();
  return (
    <button disabled={pending}>
      {pending ? 'Saving' : 'Store'} Todo
    </button>
  );
}

function App() {
  async function storeTodoAction(formData) {
    const todo = { title: formData.get('title') };
    await saveTodo(todo);
  }

  return (
    <form action={storeTodoAction}>
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" name="title" />
      </p>
      <p className="actions">
        <SubmitButton />
      </p>
    </form>
  );
}
```

In this example, the actual code to send the to-do to a backend server is extracted into a separate `saveTodo()` function that's stored in a `todo.js` file. That function contains the same code that was shown in earlier examples (i.e., it sends an HTTP request to JSONPlaceholder). In addition, `useActionState()` is removed to make the code a bit shorter and simpler again. However, you can absolutely use `useActionState()` in conjunction with `useFormStatus()`. For example, you could use `useActionState()` to output validation errors while managing the submit button's `disabled` state via `useFormStatus()` in a separate, nested component.

`useFormStatus()` is imported from `react-dom` and called inside the `SubmitButton` component function. It returns an object that contains a `pending` property yielding a boolean value.

As mentioned before, `useFormStatus()` can't be used in the component where the `<form>` element is rendered. Instead, it must be used in a nested component—that's why the `<SubmitButton>` component is placed between the `<form>` tags.

Besides `pending`, the object returned by `useFormStatus()` also holds three other properties:

- `data`: A `FormData` object that contains the data with which the parent `<form>` was submitted (i.e., the same kind of data that the form action function receives).
- `method`: A string value that's either `'get'` or `'post'`, reflecting the value to which the method prop on the `<form>` element was set. By default, it's `'get'`.
- `action`: A pointer to the form action function that's connected to the `<form>`.

If you only care about the pending status, you can, of course, either use `useActionState()` or `useFormStatus()`. Working with `useActionState()` has the advantage that no separate nested component must be built. On the other hand, creating such an extra component and relying on `useFormStatus()` might be useful if you have multiple forms on the page—you could then, for example, reuse the `<SubmitButton>` across all those forms.

Performing Optimistic Updates

Besides `useActionState()` and `useFormStatus()`, React offers one last important Hook related to forms and form actions: the `useOptimistic()` Hook.

The idea behind this Hook is that you can use it to show some temporary, optimistic UI while an asynchronous form action (which may take a couple of seconds) is underway. "Optimistic" means that you can use this Hook to render a UI that would normally only exist after the form submission is finished (e.g., a list of to-dos that already includes the newly submitted to-do).

The following example code manages a to-do list with the help of a `<form>` and form action, but without using `useOptimistic()`:

```
import { useFormStatus } from 'react-dom';

import { useState } from 'react';

let storedTodos = [];

export async function saveTodo(todo) {
  // dummy function that simulates a slow backend which manages todos
  await new Promise((resolve) => setTimeout(resolve, 3000));
  const newTodo = { ...todo, id: new Date().getTime() };
  storedTodos = [...storedTodos, newTodo];
  return storedTodos;
}

function SubmitButton() {
  // same as before, didn't change, hence omitted here
}
```

```
function App() {
  const [todos, setTodos] = useState(storedTodos);

  async function storeTodoAction(formData) {
    const todo = { title: formData.get('title') };
    const updatedTodos = await saveTodo(todo); // takes 3s
    setTodos(updatedTodos);
  }

  return (
    <>
      <form action={storeTodoAction}>
        <p>
          <label htmlFor="title">Title</label>
          <input type="text" id="title" name="title" />
        </p>
        <p className="actions">
          <SubmitButton />
       </p>
      </form>
      <div id="todos">
        <h2>My Todos</h2>
        {todos.length === 0 && <p>No todos found.</p>}
        {todos.length > 0 && (
          <ul>
            {todos.map((todo) => (
              <li key={todo.id}>{todo.title}</li>
            ))}
          </ul>
        )}
      </div>
    </>
  );
}
```

In this example, since the `saveTodo()` function again has a built-in deliberate delay of three seconds, the website user sees the outdated to-do list until the form submission process is completed.

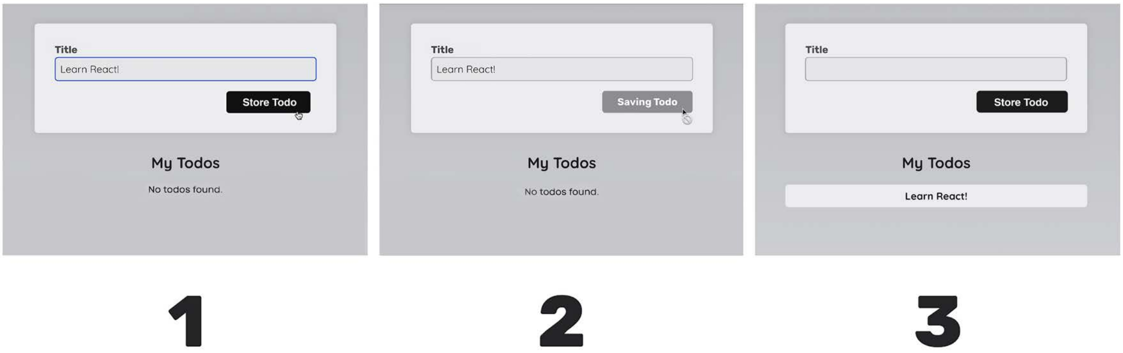


Figure 9.4: Without optimistic updating, the UI updates are delayed

The user experience can, therefore, be improved by introducing the `useOptimistic()` Hook.

This Hook requires two arguments and returns an array with exactly two elements:

```
const [optimisticState, addOptimistic] = useOptimistic(
  state, updateFunction
);
```

- `state` (the first argument) is the component state that should be active initially or if no form action is pending.
- `updateFunction` (the second argument) is a function defined by you that controls how the state should be updated optimistically.
- `optimisticState` is the optimistically updated state that will be active during the form action execution.
- `addOptimistic` triggers `updateFunction` and allows you to pass a value to that function.

Applied to the above example, `useOptimistic()` can be used to manage an alternative, optimistically updated to-dos array that will be active as long as the form action is executing. Thereafter, the regular state will be active again (and update the UI accordingly):

```
import { useOptimistic } from 'react';

import { saveTodo, getTodos } from './todos.js';
import { useState } from 'react';

function SubmitButton() {
  // same code as before, hence omitted
}

function App() {
    const loadedTodos = getTodos(); // initial fetch
    const [todos, setTodos] = useState(loadedTodos);

    const [optimisticTodos, addOptimisticTodo] = useOptimistic(
      todos,
      (currentState, optimisticValue) => {
        return [...currentState, { ...optimisticValue, id: 'temp' }];
      }
    );

    async function storeTodoAction(formData) {
      const todo = { title: formData.get('title') };
      addOptimisticTodo(todo);
      const updatedTodos = await saveTodo(todo);
      setTodos(updatedTodos);
    }

    return (
      <form action={storeTodoAction}>
        <p>
          <label htmlFor="title">Title</label>
          <input type="text" id="title" name="title" />
        </p>
        <p className="actions">
          <SubmitButton />
        </p>
      </form>
      <div id="todos">
        <h2>My Todos</h2>
        {optimisticTodos.length === 0 && <p>No todos found.</p>}
        {optimisticTodos.length > 0 && (
          <ul>
            {optimisticTodos.map((todo) => (
              <li key={todo.id}>{todo.title}</li>
            ))}
          </ul>
        )}
      </div>
    );
  }
```

As you can see in this example, the `optimisticTodos` state is now used in the JSX code. The value stored in that constant is either the normal `todos` state (managed by `useState()`), if the `storeTodoAction()` form action is not executing, or it's the array derived by the function passed to `useOptimistic()` (as the second argument).

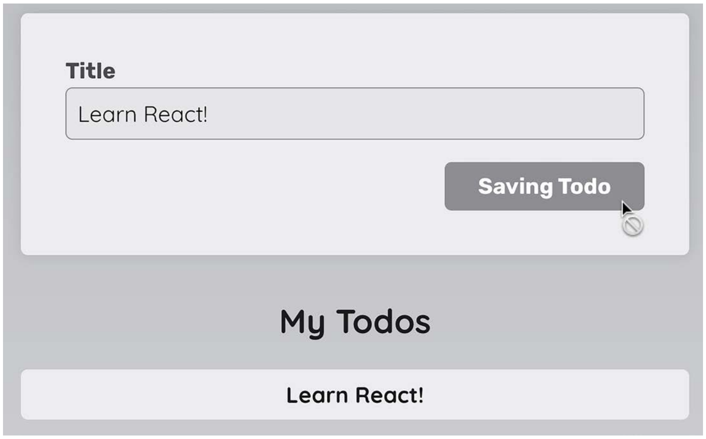


Figure 9.5: With useOptimistic(), the UI updates right away after submission

Using the `useOptimistic()` Hook can, therefore, help with building a great user experience where your application provides instant feedback, even if some slow processes might still be running in the background. Since the temporary optimistic state will always be replaced with the regular state (i.e., the `todos` state) once the form submission is done, there also is no risk of displaying an incorrect UI. If an operation fails, React will automatically replace the temporarily incorrect UI with the correct one when it falls back to using the regular state.

Summary and Key Takeaways

- Form submissions can be handled by manually listening to the `submit` event via the `onSubmit` prop.
- Alternatively, form actions can be used—i.e., functions bound to the `action` prop of a `<form>` element.
- When handling form submission manually (via `onSubmit`), you can extract form field values with the help of state (`useState()`), Refs (`useRef()`), or by creating a `FormData` object from `event.currentTarget`.
- When using form actions, a form data object with the form field input values is automatically passed to the action function as a parameter.
- The `useActionState()` Hook can be utilized to manage form-dependent state (e.g., validation error messages).
- `useActionState()` also provides a pending boolean value that may be used to update the UI while the form action is processing.
- In nested components (nested within `<form>`), the `useFormStatus()` Hook can be called to get and use information about the parent form submission status.

- To provide quick UI updates, even when dealing with slow background processes (e.g., slow HTTP requests), the `useOptimistic()` Hook may help.

What's Next?

Dealing with forms and handling user input is a very common task in most web applications. React apps are, of course, no exception.

That's why React offers a broad variety of approaches and possible patterns you can use to handle form submissions and extract user input. This chapter explored and compared the two main ways of doing this: using the `onSubmit` prop or relying on form actions (only available since React 19).

As explained and shown throughout this chapter, both approaches are valid and have their use cases. Personal preference as well as application requirements matter and will influence your decision.

At this point in the book, you know all the key React concepts you need to build feature-rich web applications. The next chapter will look behind the scenes of React and explore how it works internally. You will also learn about some common optimization techniques that can make your apps more performant.

Test Your Knowledge!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to examples that can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/09-form-actions/exercises/questions-answers.md`:

1. What's a "form action"?
2. How can you access user input inside of a form action?
3. What's the purpose of the `useActionState()` Hook and how is it used?
4. What's the purpose of the `useFormStatus()` Hook and how is it used?
5. What's the difference between `useActionState()` and `useFormStatus()`?
6. What's the purpose of the `useOptimistic()` Hook and how is it used?

Apply What You Learned

With form actions in your React toolbelt, you have another powerful way of handling form submissions and extracting user input.

In the following section, you'll find an activity that allows you to practice working with form actions and the form-related Hooks provided by React. As always, you will also need to employ some of the concepts covered in earlier chapters (such as working with state or outputting lists).

Activity 9.1: Managing a Feedback Form

In this activity, your job is to build upon an existing, basic feedback form application and handle form submissions, with the help of form actions. As part of this activity, you should validate the submitted title and feedback text and show error messages if empty values are submitted. You should also update the list of submitted feedback items optimistically and disable the submit button while the form action is underway.

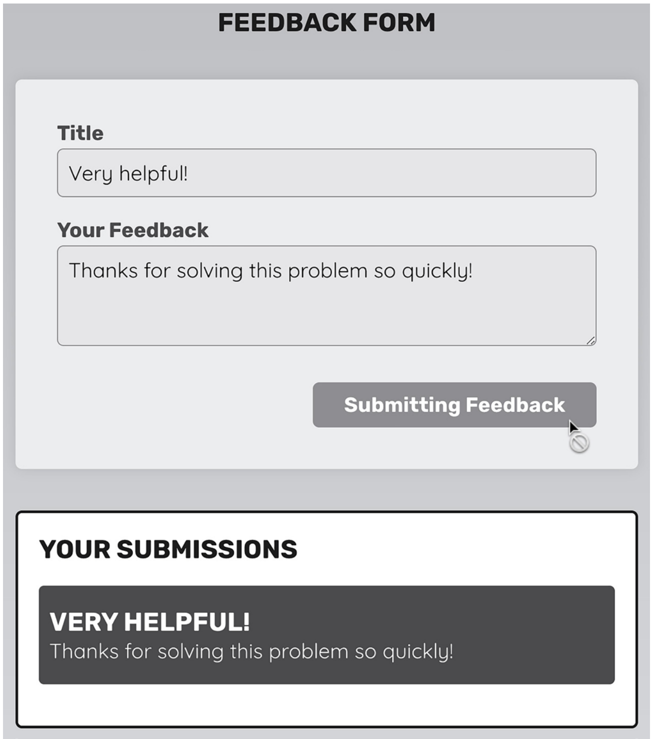
Note

You can find the starting code for this activity at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/09-form-actions/activities/practice-1-start`. When downloading this code, you'll always download the entire repository. Make sure to then navigate to the subfolder with the starting code (`activities/practice-1-start`, in this case) to use the right code snapshot.

After downloading the code and running `npm install` in the project folder to install all required dependencies, the solution steps are as follows:

1. Replace the existing `onSubmit` handler function with a form action—clean up and remove any code that's not needed anymore thereafter.
2. Disable the form submit button while the form action is processing.
3. Validate the user input and output any error messages with the help of the `useActionState()` Hook.
4. Update the list of submitted feedback items optimistically by utilizing the `useOptimistic()` Hook.

The expected result should resemble the following screenshots:



FEEDBACK FORM

Title

Very helpful!

Your Feedback

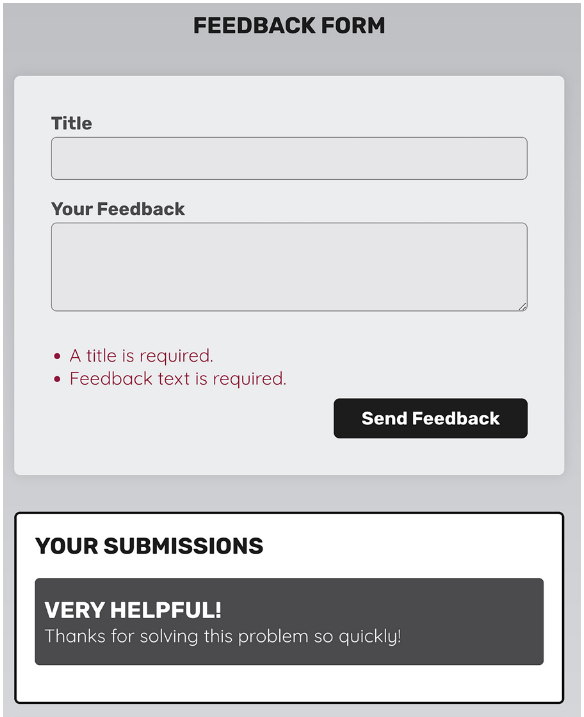
Thanks for solving this problem so quickly!

YOUR SUBMISSIONS

VERY HELPFUL!

Thanks for solving this problem so quickly!

Figure 9.6: During form submission, the button is disabled, but the submitted item shows up instantly



FEEDBACK FORM
Title
Your Feedback
Send Feedback
YOUR SUBMISSIONS

- A title is required.
- Feedback text is required.

VERY HELPFUL!
Thanks for solving this problem so quickly!

Figure 9.7: When submitting invalid values, appropriate error messages are shown

Note

You can find a full example solution here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/09-form-actions/activities/practice-1`.

10

Behind the Scenes of React and Optimization Opportunities

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Avoid unnecessary code execution via the `useMemo()` and `useCallback()` Hooks
- Load optional code lazily, only when it's needed, via React's `lazy()` function
- Use React's developer tools to analyze and optimize your app
- Explore the React Compiler for automatic performance improvements

Introduction

Using all the features covered up to this point, you can build non-trivial React apps and therefore highly interactive and reactive UIs.

This chapter, while introducing some new functions and concepts, will not provide you with tools that will enable you to build even more advanced web apps. You will not learn about groundbreaking, key concepts such as state or props (though you will learn about more advanced concepts in later chapters).

Instead, this chapter allows you to look behind the scenes of React. You will learn how React calculates required DOM updates, and how it ensures that such updates happen without impacting performance in an unacceptable way. You will also learn about some other optimization techniques employed by React—all with the goal of ensuring that your React app runs as smoothly as possible.

Besides this look behind the scenes, you will learn about various built-in functions and concepts that can be used to further optimize app performance. This chapter will not only introduce those concepts but also explain **why** they exist, **how** they should be used, and **when** to use which feature.

Revisiting Component Evaluations and Updates

Before exploring React's internal workings, it makes sense to briefly revisit React's logic for executing component functions.

Component functions are executed whenever some state (managed via `useState()`) changes or their parent component function is executed again. The latter happens because, if a parent component function is called, its entire JSX code (which points at the child component function) is re-evaluated. Any component functions referenced in that JSX code are therefore also invoked again.

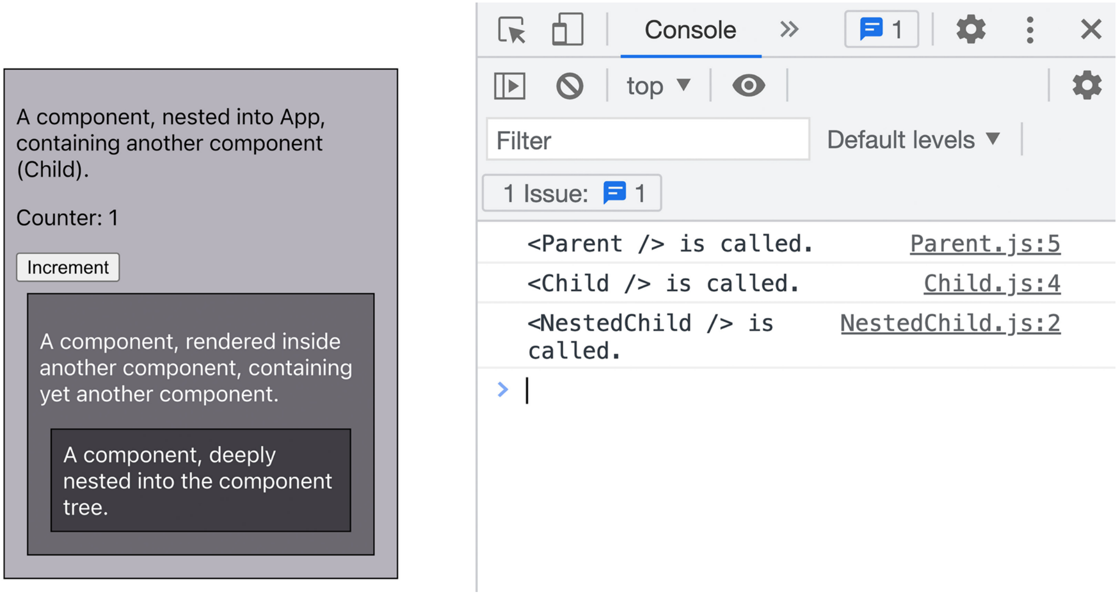
Consider a component structure like this:

```
function NestedChild() {
  console.log('<NestedChild /> is called.');

  return (
    <p id="nested-child">
      A component, deeply nested into the component tree.
    </p>
  );
}

function Child() {
  console.log('<Child /> is called.');

  return (
    <div id="child">
      <p>
        A component, rendered inside another component,
        containing yet another component.
      </p>
      <NestedChild />
    </div>
  );
}

function Parent() {
  console.log('<Parent /> is called.');

  const [counter, setCounter] = useState(0);

  function handleIncCounter() {
    setCounter((prevCounter) => prevCounter + 1);
  }

  return (
    <div id="parent">
      <p>
        A component, nested into App,
        containing another component (Child).
      </p>
      <p>Counter: {counter}</p>
      <button onClick={handleIncCounter}>Increment</button>
      <Child />
    </div>
  );
}
```

In this example structure, the `Parent` component renders a `<div>` with two paragraphs, a button, and another component: the `Child` component. That component in turn outputs a `<div>` with a paragraph and yet another component: the `NestedChild` component (which then only outputs a paragraph).

The `Parent` component also manages some state (a dummy counter), which is changed whenever the button is clicked. All three components print a message via `console.log()`, simply to make it easy to spot when each component is called by React.

The following screenshot shows those components in action—after the button was clicked:


A component, nested into App, containing another component (Child).

Counter: 1

A component, rendered inside another component, containing yet another component.

A component, deeply nested into the component tree.

```
<Parent /> is called.        Parent.js:5
<Child /> is called.          Child.js:4
<NestedChild /> is      NestedChild.js:2
called.
```

Figure 10.1: Each component function is executed

In this screenshot, you can not only see how the components are nested into each other but also how they are all invoked by React when the `Increment` button is clicked. `Child` and `NestedChild` are invoked even though they don't manage or use any state. But since they are a child (`Child`) or descendent (`NestedChild`) of the `Parent` component, which did receive a state change, the nested component functions are called as well.

Understanding this flow of component function execution is important because this flow implies that any component function invocation also influences its descendent components. It also shows you how frequently component functions may be invoked by React and how many component functions may be affected by a single state change.

Therefore, there's one important question that should be answered: what happens to the actual page DOM (i.e., to the loaded and rendered website in the browser) when one or more component functions are invoked? Is the DOM recreated? Is the rendered UI updated?

What Happens When a Component Function Is Called

Whenever a component function is executed, React evaluates whether or not the rendered UI (i.e., the DOM of the loaded page) must be updated.

This is important: React **evaluates** whether an update is needed. It's not forcing an update automatically!

Internally, React does not take the JSX code returned by a component (or multiple components) and replace the page DOM with it.

That could be done, but it would mean that every component function execution would lead to some form of DOM manipulation—even if it's just a replacement of the old DOM content with a new, similar version. In the example shown above, the `Child` and `NestedChild` JSX code would be used to replace the currently rendered DOM every time those component functions were executed.

As you can see in the example above, those component functions are executed quite frequently. But the returned JSX code is always the same because it's static. It does not contain any dynamic values (e.g., state or props).

If the actual page DOM were replaced with the DOM elements implied by the returned JSX code, the visual result would always be the same. But there still would be some DOM manipulation behind the scenes. And that's a problem because manipulating the DOM is quite a performance-intensive task—especially when done with a high frequency. Removing and adding or updating DOM elements should therefore only be done when needed—not unnecessarily.

Because of this, React does not throw away the current DOM and replace it with the new DOM (implied by the JSX code) just because a component function was executed. Instead, React first checks whether an update is needed. And if it's needed, only the parts of the DOM that need to change are replaced or updated.

To determine whether an update is needed (and where), React uses a concept called the **virtual DOM**.

The Virtual DOM vs the Real DOM

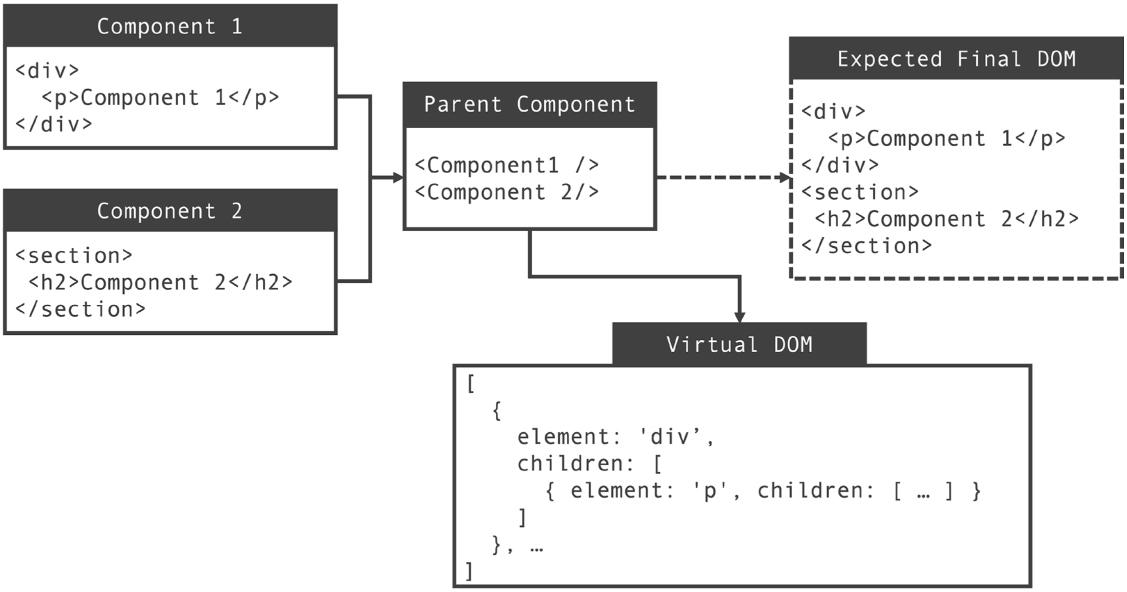
To determine whether (and where) a DOM update might be needed, React (specifically, the `react-dom` package) compares the current DOM structure to the structure implied by the JSX code returned by the executed component functions. If there's a difference, the DOM is updated accordingly; otherwise, it's left untouched.

However, just as manipulating the DOM is relatively performance-intensive, reading the DOM is as well. Even without changing anything in the DOM, reaching out to it, traversing the DOM elements, and deriving the structure from it is something you typically want to reduce to a minimum.

If multiple component functions are executed and each triggers a process where the rendered DOM elements are read and compared to the JSX structure implied by the invoked component functions, the rendered DOM will be hit with read operations multiple times within a very short time frame.

For bigger React apps that are made up of dozens, hundreds, or even thousands of components, it's highly probable that dozens of component function executions might occur within a single second. If that were to lead to the same amount of DOM read operations, there's a quite high chance that the web app would feel slow or laggy to the user.

That's why React does not use the real DOM to determine whether any UI updates are needed. Instead, it constructs and manages a virtual DOM internally—an in-memory representation of the DOM that's rendered in the browser. The virtual DOM is not a browser feature, but a React feature. You can think of it as a deeply nested JavaScript object that reflects the components of your web app, including all the built-in HTML components such as `<div>`, `<p>`, etc. (that is, the actual HTML elements that should show up on the page in the end).



```
<div>
  <p>Component 1</p>
</div>
```

```
<section>
 <h2>Component 2</h2>
</section>
```

```
<Component1 />
<Component 2/>
```

```
<div>
  <p>Component 1</p>
</div>
<section>
 <h2>Component 2</h2>
</section>
```

```
[
  {
    element: 'div',
    children: [
      { element: 'p', children: [ … ] }
    ]
  }, …
]
```

Figure 10.2: React manages a virtual representation of the expected element structure

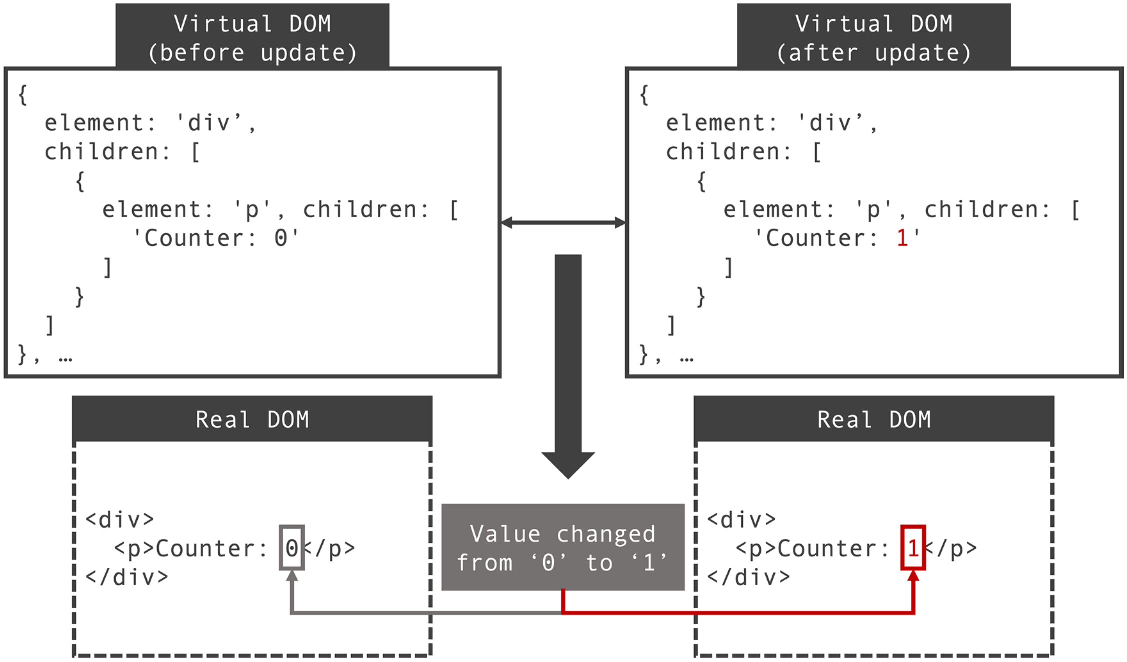
In the figure above, you can see that the expected element structure (in other words, the expected final DOM) is actually stored as a JavaScript object (or an array with a list of objects). This is the virtual DOM, which is managed by React and used for identifying required DOM updates.

Note

Please note that the actual structure of the virtual DOM is more complex than the structure shown in the image. The chart above aims to give you an idea of what the virtual DOM is and how it might look. It's not an exact technical representation of the JavaScript data structure managed by React.

React manages this virtual DOM because comparing this virtual DOM to the expected UI state is much less performance-intensive than reaching out to the real DOM.

Whenever a component function is called, React compares the returned JSX code to the respective virtual DOM nodes stored in the virtual DOM. If differences are detected, React will determine which changes are needed to update the DOM. Once the required adjustments are derived, these changes are applied to both the virtual and the real DOM. This ensures that the real DOM reflects the expected UI state without having to reach out to it or update it all the time.



Virtual DOM (before update)

```
{
  element: 'div',
  children: [
    {
      element: 'p', children: [
        'Counter: 0'
      ]
    }
  ]
}, …
```

Virtual DOM (after update)

```
{
  element: 'div',
  children: [
    {
      element: 'p', children: [
        'Counter: 1'
      ]
    }
  ]
}, …
```

```
<div>
  <p>Counter: 0</p>
</div>
```

Value changed from '0' to '1'

```
<div>
  <p>Counter: 1</p>
</div>
```

Figure 10.3: React detects required updates via the virtual DOM

In the figure above, you can see how React compares the current DOM and the expected structure with the help of the virtual DOM first, before reaching out to the real DOM to manipulate it accordingly.

As a React developer, you don't need to actively interact with the virtual DOM. Technically, you don't even need to know that it exists and that React uses it internally. But it's always helpful to understand the tool (React, in this case) you're working with. It's good to know that React does various performance optimizations for you and that you get those on top of the many other features that make your life as a developer (hopefully) easier.

State Batching

Since React uses the concept of a virtual DOM, frequent component function executions aren't a huge problem. But of course, even if comparisons are only conducted virtually, there is still some internal code that must be executed. Even with the virtual DOM, performance could degrade if lots of unnecessary (and at the same time quite complex) virtual DOM comparisons must be made.

One scenario where unnecessary comparisons are performed is in the execution of multiple sequential state updates. Since each state update causes the component function to be executed again (as well as all potential nested components), multiple state updates that are performed together (for example, in the same event handler function) will cause multiple component function invocations.

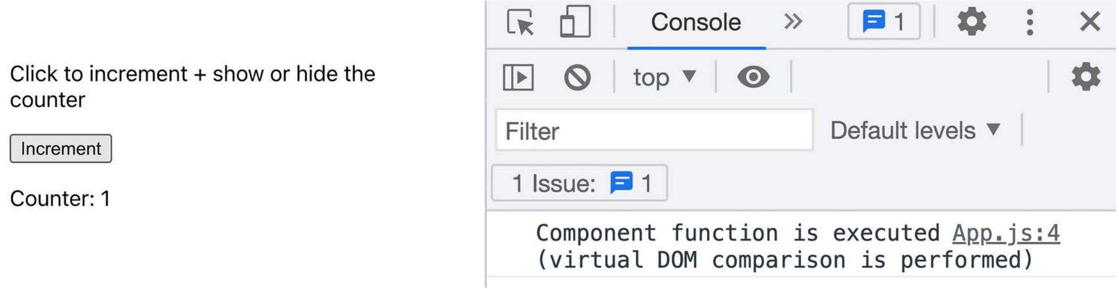
Consider this example:

```
function App() {
  const [counter, setCounter] = useState(0);
  const [showCounter, setShowCounter] = useState(false);

  function handleIncCounter() {
    setCounter((prevCounter) => prevCounter + 1);
    setShowCounter(true);
  }

  return (
    <>
      <p>Click to increment + show or hide the counter</p>
      <button onClick={handleIncCounter}>Increment</button>
      {showCounter && <p>Counter: {counter}</p>}
    </>
  );
}
```

This component contains two state values: `counter` and `showCounter`. When the button is clicked, the counter is incremented by `1`. In addition, `showCounter` is set to `true`. Therefore, the first time the button is clicked, both the `counter` and the `showCounter` states are changed (because `showCounter` is `false` initially).

Since two state values are changed, the expectation would be that the `App` component function is called twice by React—because every state update causes the connected component function to be invoked again.

However, if you add a `console.log()` statement to the `App` component function (to track how often it's executed), you will see that it's only invoked once, when the `Increment` button is clicked:



Click to increment + show or hide the counter

Counter: 1

```
Component function is executed App.js:4
(virtual DOM comparison is performed)
```

Figure 10.4: Only one console log message is displayed

Note

If you see two log messages instead of one, make sure you're not using React's "Strict Mode." When running in Strict Mode during development, React executes component functions more often than it normally would.

If necessary, you can disable Strict Mode by removing the `<React.StrictMode>` component from your `main.jsx` file. You will learn more about React's Strict Mode toward the end of this chapter.

This behavior is called **state batching**. React performs state batching when multiple state updates are initiated from the same place in your code (e.g., from inside the same event handler function).

It's a built-in functionality that ensures that your component functions are not called more often than needed. This prevents unnecessary virtual DOM comparisons.

State batching is a very useful mechanism. But there is another kind of unnecessary component evaluation that it does not prevent: child component functions that get executed when the parent component function is called.

Avoiding Unnecessary Child Component Evaluations

Whenever a component function is invoked (because its state changed, for example), any nested component functions will be called as well. See the first section of this chapter for more details.

As you saw in the example in the first section of this chapter, it is often the case that those nested components don't actually need to be evaluated again. They might not depend on the state value that changed in the parent component. They might not even depend on any values of the parent component at all.

Here's an example where the parent component function contains some state that is not used by the child component:

```
function Error({ message }) {
  if (!message) {
    return null;
  }

  return <p className={classes.error}>{message}</p>;
}


function Form() {
  const [enteredEmail, setEnteredEmail] = useState('');
  const [errorMessage, setErrorMessage] = useState();

  function handleUpdateEmail(event) {
    setEnteredEmail(event.target.value);
  }

  function handleSubmit(event) {
    event.preventDefault();
    if (!enteredEmail.endsWith('.com')) {
      setErrorMessage('Email must end with .com.');
    }
  }

  return (
    <form className={classes.form} onSubmit={handleSubmit}>
      <div className={classes.control}>
        <label htmlFor="email">Email</label>
        <input
          id="email"
          type="email"
          value={enteredEmail}
          onChange={handleUpdateEmail}
        />
      </div>
      <Error message={errorMessage} />
      <button>Sign Up</button>
    </form>
  );
}
```

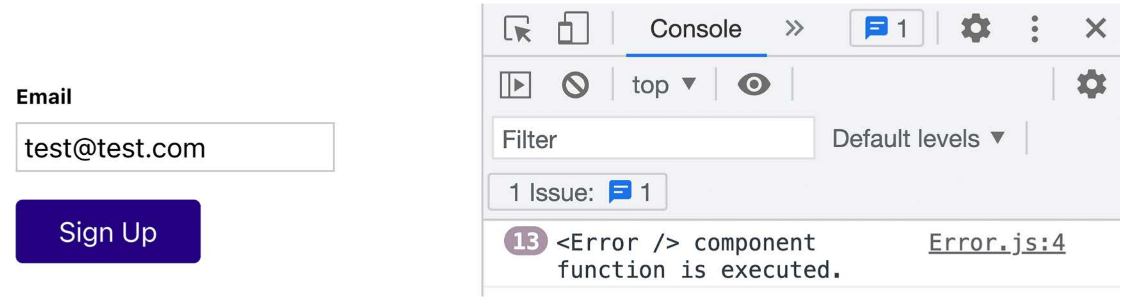
Note

You can find the complete example code on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/10-behind-scenes/examples/03-memo`.

In this example, the `Error` component relies on the `message` prop, which is set to the value stored in the `errorMessage` state of the `Form` component. However, the `Form` component also manages an `enteredEmail` state, which is not used (not received via props) by the `Error` component. Therefore, changes to the `enteredEmail` state will cause the `Error` component to be executed again, despite the component not needing that value.

You can track the unnecessary `Error` component function invocations by adding a `console.log()` statement to that component function:

```
function Error({ message }) {
  console.log('<Error /> component function is executed.');
  if (!message) {
    return null;
  }

  return <p className={classes.error}>{message}</p>;
}
```



Email

test@test.com

Sign Up

```
13 <Error /> component function is executed.   Error.js:4
```

Figure 10.5: The Error component function is executed for every keystroke

In the preceding screenshot, you can see that the `Error` component function is executed for every keystroke on the input field (that is, once for every `enteredEmail` state change).

This is in line with what you have learned previously, but it is also unnecessary. The `Error` component does depend on the `errorMessage` state and should certainly be re-evaluated whenever that state changes, but executing the `Error` component function because the `enteredEmail` state value was updated is clearly not required.

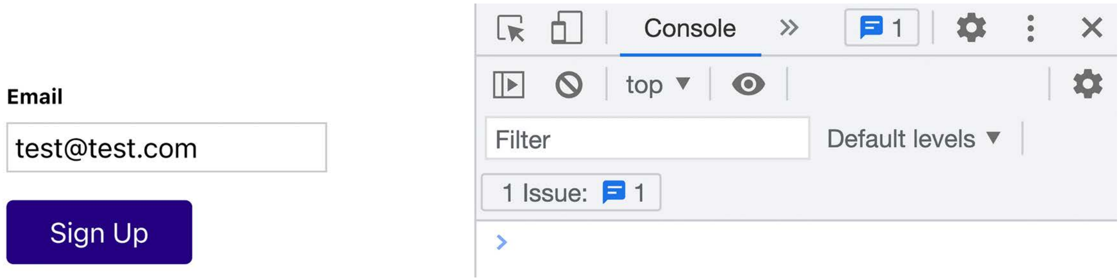
That's why React offers another built-in function that you can use to control (and prevent) this behavior: the `memo()` function.

`memo` is imported from `react` and is used like this:

```
import { memo } from 'react';

import classes from './Error.module.css';

function Error({ message }) {
  console.log('<Error /> component function is executed.');
  if (!message) {
    return null;
  }

  return <p className={classes.error}>{message}</p>;
}

export default memo(Error);
```

You wrap the component function that should be protected from unnecessary, parent-initiated re-evaluations with `memo()`. This causes React to check whether the component's props did change, compared to the last time the component function was called. If prop values are equal, React knows that the component function does not need to be executed again.

By adding `memo()`, the unnecessary component function invocations are avoided, as shown below:



test@test.com

Figure 10.6: No console log messages appear

As you can see in the figure, no messages are printed to the console. This proves that unnecessary component executions are avoided (remember: before adding `memo()`, many messages were printed to the console).

`memo()` also takes an optional second argument that can be used to add your own logic to determine whether prop values have changed or not. This can be useful if you're dealing with more complex prop values (e.g., objects or arrays) where custom comparison logic might be needed, as in the following example:

```
memo(SomeComponent, function(prevProps, nextProps) {
  return prevProps.user.firstName !== nextProps.user.firstName;
});
```

The (optional) second argument passed to `memo()` must be a function that automatically receives the previous props object and the next props object. The function then must return `true` if the component (`SomeComponent`, in this example) should be re-evaluated and `false` if it should not.

Often, the second argument is not needed because the default behavior of `memo()` (where it compares all props for inequality) is exactly what you need. But if more customization or control is needed, `memo()` allows you to add your custom logic.

With `memo()` in your toolbox, it's tempting to wrap every React component function with `memo()`. Why wouldn't you do it? After all, it avoids unnecessary component function executions.

You definitely can use it on all components—but it's not necessarily helpful to do that because avoiding unnecessary component re-evaluations by using `memo()` comes at a cost: comparing props (old versus new) also requires some code to run. It's not "free." It's not a huge cost, though. Using `memo()` on many (or all) components will likely not slow down your app significantly. But it's still unnecessary if you have components that do need to be re-evaluated a lot. Using `memo()` on components that receive props that change a lot simply does not do anything useful.

Hence `memo()` makes the most sense if you have relatively simple props (i.e., props with no deeply nested objects that you need to compare manually with a custom comparison function) and most parent component state changes don't affect those props of the child component. And even in those cases, if you have a relatively simple component function (i.e., without any complex logic in it), using `memo()` still might not yield any measurable benefit.

The example code above (the `Error` component) is a good example: in theory, using `memo()` makes sense here. Most state changes in the parent component won't affect `Error`, and the prop comparison will be very simple because it's just one prop (the `message` prop, which holds a string) that must be compared. But despite that, using `memo()` to wrap `Error` will very likely not be worth it. `Error` is an extremely basic component with no complex logic in it. It simply doesn't matter if the component function gets invoked frequently. Hence, using `memo()` in this spot would be absolutely fine—but so is not using it.

A great spot to use `memo()`, on the other hand, is a component that's relatively close to the top of the component tree (or of a deeply nested branch of components in the component tree). If you are able to avoid unnecessary executions of that one component via `memo()`, you'll also implicitly avoid unnecessary executions of all nested components beneath that one component. This is illustrated in the diagram below:

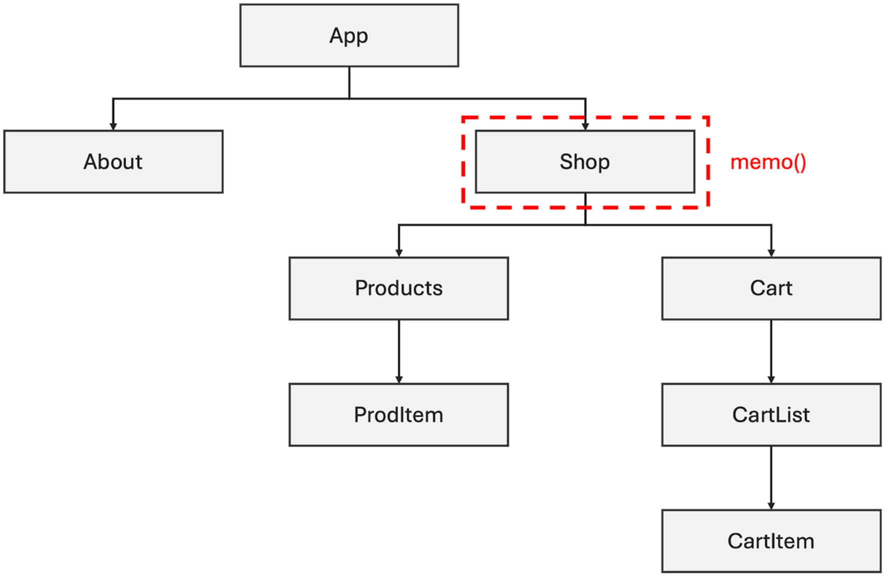


Figure 10.7: Using memo at the start of a component tree branch

In the preceding figure, `memo()` is used on the `Shop` component, which has multiple nested descendent components. Without `memo()`, whenever the `Shop` component function gets invoked, `Products`, `ProdItem`, `Cart`, etc. would also be executed. With `memo()`, assuming that it's able to avoid some unnecessary executions of the `Shop` component function, all those descendent components are no longer evaluated.

Avoiding Costly Computations

The `memo()` function can help avoid unnecessary component function executions. As mentioned in the previous section, this is especially valuable if a component function performs a lot of work (e.g., sorting a long list).

But as a React developer, you will also encounter situations in which you have a work-intensive component that needs to be executed again because some prop value changed. In such cases, using `memo()` won't prevent the component function from executing again. However, the prop that changed might not be needed for the performance-intensive task that is performed as part of the component.

Consider the following example:

```
function sortItems(items) {
  console.log('Sorting');
  return items.sort(function (a, b) {
    if (a.id > b.id) {
      return 1;
    } else if (a.id < b.id) {
      return -1;
    }
    return 0;
  });
}

function List({ items, maxNumber }) {
  const sortedItems = sortItems(items);

  const listItems = sortedItems.slice(0, maxNumber);

  return (
    <ul>
      {listItems.map((item) => (
        <li key={item.id}>
          {item.title} (ID: {item.id})
        </li>
      ))}
    </ul>
  );
}

export default List;
```

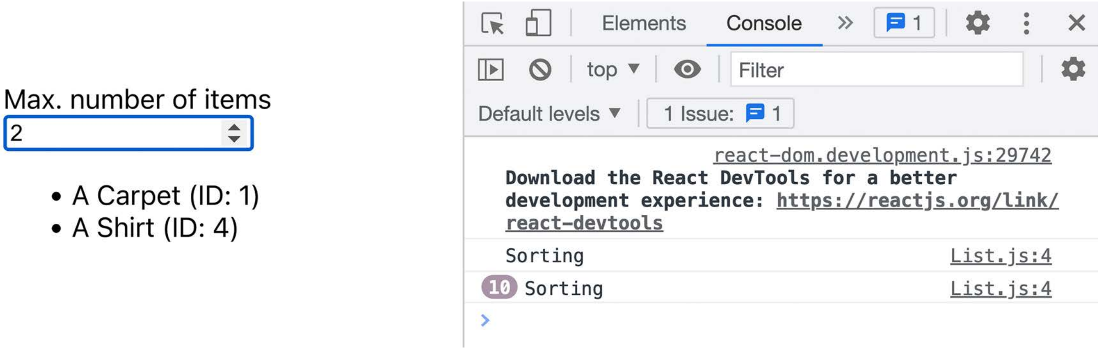
The `List` component receives two prop values: `items` and `maxNumber`. It then calls `sortItems()` to sort the items by `id`. Thereafter, the sorted list is limited to a certain amount (`maxNumber`) of items. As a last step, the sorted and shortened list is then rendered to the screen via `map()` in the JSX code.

Note

A full example app can be found on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/10-behind-scenes/examples/04-usememo`.

Depending on the number of items passed to the `List` component, sorting it can take a significant amount of time (for very long lists, even up to a few seconds). It's definitely not an operation you want to perform unnecessarily or too frequently. The list needs to be sorted whenever `items` changes, but it should not be sorted if `maxNumber` changes—because this does not impact the items in the list (i.e., it doesn't affect the order). But with the code snippet shared above, `sortItems()` will be executed whenever either of the two prop values changes, no matter whether it's `items` or `maxNumber`.

As a result, when running the app and changing the number of displayed items, you can see multiple `"Sorting"` log messages—implying that `sortItems()` was executed every time the number of items was changed.



Max. number of items
2

- A Carpet (ID: 1)
- A Shirt (ID: 4)

```
react-dom.development.js:29742
Download the React DevTools for a better
development experience: https://reactjs.org/link/
react-devtools
Sorting                                     List.js:4
10 Sorting                                  List.js:4
```

Figure 10.8: Multiple "Sorting" log messages appear in the console

The `memo()` function won't help here because the `List` component function should (and will) execute whenever `items` or `maxNumber` changes. `memo()` does not help control partial code execution inside the component function.

For that, you can use another feature provided by React: the `useMemo()` Hook.

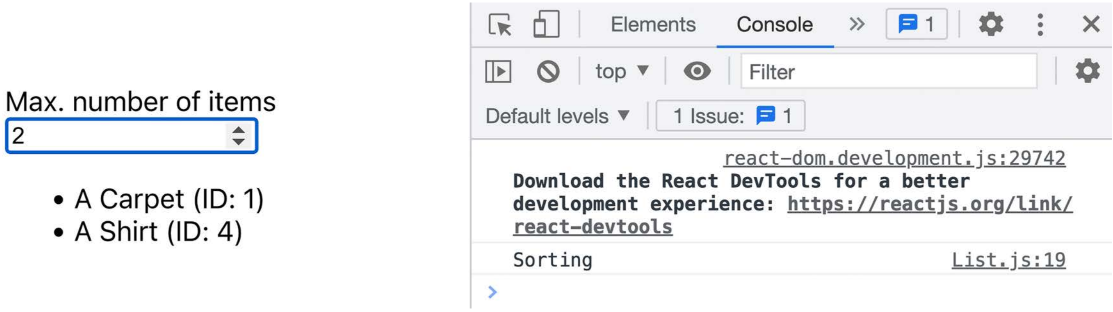
`useMemo()` can be used to wrap a compute-intensive operation. For it to work correctly, you also must define a list of dependencies that should cause the operation to be executed again. To some extent, it's similar to `useEffect()` (which also wraps an operation and defines a list of dependencies), but the key difference is that `useMemo()` runs at the same time as the rest of the code in the component function, whereas `useEffect()` executes the wrapped logic after the component function execution has finished. `useEffect()` should not be used for optimizing compute-intensive tasks but for side effects.

`useMemo()`, on the other hand, exists to control the execution of performance-intensive tasks. Applied to the example mentioned above, the code can be adjusted like this:

```
import { useMemo } from 'react';

function List({ items, maxNumber }) {
  const sortedItems = useMemo(
    function() {
      console.log('Sorting');
      return items.sort(function (a, b) {
        if (a.id > b.id) {
          return 1;
        } else if (a.id < b.id) {
          return -1;
        }
        return 0;
      });
    },
    [items]
  );

  const listItems = sortedItems.slice(0, maxNumber);

  return (
    <ul>
      {listItems.map((item) => (
        <li key={item.id}>
          {item.title} (ID: {item.id})
        </li>
      ))}
    </ul>
  );
}

export default List;
```

`useMemo()` wraps an anonymous function (the function that previously existed as a named function, `sortItems`), which contains the entire sorting code. The second argument passed to `useMemo()` is the array of dependencies for which the function should be executed again (when a dependency value changes). In this case, `items` is the only dependency of the wrapped function, and so that value is added to the array.

With `useMemo()` used like this, the sorting logic will only execute when items change, not when `maxNumber` (or anything else) changes. As a result, you see `"Sorting"` being output in the developer tools console only once:



Max. number of items
2

- A Carpet (ID: 1)
- A Shirt (ID: 4)

```
                    react-dom.development.js:29742
Download the React DevTools for a better
development experience: https://reactjs.org/link/
react-devtools
Sorting                                 List.js:19
>
```

Figure 10.9: Only one "Sorting" output in the console

`useMemo()` can be very useful for controlling code execution inside of your component functions. It can be a great addition to `memo()` (which controls the overall component function execution). But, also like `memo()`, you should not start wrapping all your logic with `useMemo()`. Only use it for very performance-intensive computations since checking for dependency changes and storing and retrieving past computation results (which `useMemo()` does internally) also comes at a performance cost.

Utilizing useCallback()

In the previous chapters, you learned about `useCallback()`. Just like `useMemo()` can be used for "expensive" calculations, `useCallback()` can be used to prevent unnecessary function re-creations. In the context of this chapter, `useCallback()` can be helpful because, in conjunction with `memo()` or `useMemo()`, it can help you avoid unnecessary code execution. It can help you in cases where a function is passed as a prop (i.e., where you might use `memo()`) or is used as a dependency in some "expensive" computation (i.e., possibly solved via `useMemo()`).

Here's an example where `useCallback()` can be combined with `memo()` to prevent unnecessary component function executions:

```
import { memo } from 'react';

import classes from './Error.module.css';

function Error({ message, onClearError }) {
  console.log('<Error /> component function is executed.');
  if (!message) {
    return null;
  }

  return (
    <div className={classes.error}>
      <p>{message}</p>
      <button className={classes.errorBtn} onClick={onClearError}>X</button>
    </div>
  );
}

export default memo(Error);
```

The `Error` component is wrapped with the `memo()` function and so will only execute if one of the received prop values changes.

The `Error` component is used by another component, the `Form` component, like this:

```
function Form() {
  const [enteredEmail, setEnteredEmail] = useState('');
  const [errorMessage, setErrorMessage] = useState();

  function handleUpdateEmail(event) {
    setEnteredEmail(event.target.value);
  }
  function handleSubmit(event) {
    event.preventDefault();
    if (!enteredEmail.endsWith('.com')) {
      setErrorMessage('Email must end with .com.');
    }
  }

  function handleClearError() {
    setErrorMessage(null);
  }

  return (
    <form className={classes.form} onSubmit={handleSubmit}>
      <div className={classes.control}>
        <label htmlFor="email">Email</label>
        <input
          id="email"
          type="email"
          value={enteredEmail}
          onChange={handleUpdateEmail}
        />
      </div>
      <Error message={errorMessage} onClearError={handleClearError} />
      <button>Sign Up</button>
    </form>
  );
}
```

In this component, the `Error` component receives a pointer to the `handleClearError` function (as a value for the `onClearError` prop). You might recall a very similar example from earlier in this chapter (from the *Avoiding Unnecessary Child Component Evaluations* section). There, `memo()` was used to ensure that the `Error` component function was not invoked when `enteredEmail` changed (because its value was not used in the `Error` component function at all).

Now, with the adjusted example and the `handleClearError` function pointer passed to `Error`, `memo()` unfortunately isn't preventing component function executions anymore. Why? Because functions are objects in JavaScript and the `handleClearError` function is recreated every time the `Form` component function is executed (which happens on every state change, including changes to the `enteredEmail` state).

Since a new function object is created for every state change, `handleClearError` is technically a different value for every execution of the `Form` component. Therefore, the `Error` component receives a new `onClearError` prop value whenever the `Form` component function is invoked. To `memo()`, the old and new `handleClearError` function objects are different from each other, and it therefore will not stop the `Error` component function from running again.

That's exactly where `useCallback()` can help:

```
const handleClearError = useCallback(() => {
  setErrorMessage(null);
}, []);
```

By wrapping `handleClearError` with `useCallback()`, the re-creation of the function is prevented, and so no new function object is passed to the `Error` component. Hence, `memo()` is able to detect equality between the old and new `onClearError` prop value and prevents unnecessary function component executions again.

Similarly, `useCallback()` can be used in conjunction with `useMemo()`. If the compute-intensive operation wrapped with `useMemo()` uses a function as a dependency, you can use `useCallback()` to ensure that this dependent function is not recreated unnecessarily.

Using the React Compiler

Considering and using `memo()`, `useMemo()`, and `useCallback()` to prevent unnecessary component re-evaluations can be a chore. Even though performance optimization is important, as a React developer, you typically want to focus on building great UIs and implementing helpful features in them.

That's why the React team developed a compiler that aims to optimize code for you – a standalone tool that can be added to React projects that will automatically wrap your components with `memo()`, use `useMemo()` when needed, and wrap functions with `useCallback()`.

Therefore, when using this compiler, you don't have to think about or use these optimization functions and Hooks anymore.

In other words, the React compiler will optimize your code for you. At least, that's the theory.

However, at the time of writing, this compiler is only available in experimental mode. This means that you shouldn't use it for production and that there may be bugs or suboptimal compilation results.

Nonetheless, you can give it a try when working on a project that uses React 19 or higher (the compiler won't work with older React versions).

Adding the compiler to a project is easy since it's just an extra dependency that must be installed in your project:

```
npm install babel-plugin-react-compiler
```

Note

Since the compiler is not stable yet, it's possible that installation steps and usage instructions will change over time.

Therefore, you should visit the official React compiler documentation page for the latest details and instructions: `https://react.dev/learn/react-compiler`.

With the compiler plugin installed, you must adjust your build process configuration such that the compiler is used. When working on a Vite-based project, you just have to edit the `vite.config.js` file, which should exist in your root project folder:

```
// vite.config.js
const ReactCompilerConfig = { /* ... */ };

export default defineConfig(() => {
  return {
    plugins: [
      react({
        babel: {
          plugins: [
            ["babel-plugin-react-compiler", ReactCompilerConfig],
          ],
        },
      }),
    ],
    // ...
  };
});
```

If you're using another project setup, you can follow the installation instructions on the official compiler documentation page.

With the compiler installed, it will automatically be executed to analyze and adjust your code to include optimizations like `memo()` or `useMemo()`. Keep in mind that those optimizations are performed as part of the build process that's invoked by running `npm run dev` or `npm run build`. Therefore, your original source code will not change—instead, the compiler optimizes your code *"behind the scenes."*

Once the React compiler is stable, it will very likely be a standard tool that's part of every React project's build process. Therefore, you won't have to use `memo()`, `useMemo()`, or `useCallback()` manually in your code anymore. But until that's the case, or in React projects that can't use the compiler, you'll still have to optimize the code manually.

Avoiding Unnecessary Code Download

Thus far, this chapter has mostly discussed strategies for avoiding unnecessary code execution. But it's not just the execution of code that can be an issue. It's also not great if your website visitors have to download lots of code that might never be executed at all. Because every kilobyte of JavaScript code that has to be downloaded will slow down the initial loading time of your web page—not just because of the time it takes to download the code bundle (which can be significant, if users are on a slow network and code bundles are big) but also because the browser has to parse all the downloaded code before your page becomes interactive.

For this reason, a lot of community and ecosystem effort is spent on reducing JavaScript code bundle sizes. Minification (automatic shortening of variable names and other measures to reduce the final code) and compression can help a lot and is therefore a common technique. Actually, projects created with Vite already come with a build workflow (initiated by running `npm run build`), which will produce a production-optimized code bundle that is as small as possible.

But there also are steps that can be taken by you, the developer, to reduce the overall code bundle size:

1. Try to write short and concise code.
2. Be thoughtful about including third-party libraries and don't use them unless you really need to.
3. Consider using code-splitting techniques.

The first point should be fairly obvious. If you write less code, your website visitors have less code to download. Therefore, trying to be concise and write optimized code makes sense.

The second point should also make sense. For some tasks, you will actually save code by including third-party libraries that may be much more elaborate than the code solution you might come up with. But there are also situations and tasks in which you might get away with writing your own code or using some built-in function instead of including a third-party library. You should at least always think about this alternative and only include third-party libraries you absolutely need.

The last point is something React can help with.

Reducing Bundle Sizes via Code Splitting (Lazy Loading)

React exposes a `lazy()` function that can be used to load component code conditionally—meaning only when it's actually needed (instead of upfront).

Consider the following example, consisting of two components working together.

A `DateCalculator` component is defined like this:

```
import { useState } from 'react';
import { add, differenceInDays, format, parseISO } from 'date-fns';

import classes from './DateCalculator.module.css';

const initialStartDate = new Date();
const initialEndDate = add(initialStartDate, { days: 1 });

function DateCalculator() {
  const [startDate, setStartDate] = useState(
    format(initialStartDate, 'yyyy-MM-dd')
  );
  const [endDate, setEndDate] = useState(
    format(initialEndDate, 'yyyy-MM-dd')
  );

  const daysDiff = differenceInDays(
    parseISO(endDate),
    parseISO(startDate)
  );

  function handleUpdateStartDate(event) {
    setStartDate(event.target.value);
  }

  function handleUpdateEndDate(event) {
    setEndDate(event.target.value);
  }

  return (
    <div className={classes.calculator}>
      <p>Calculate the difference (in days) between two dates.</p>
      <div className={classes.control}>
        <label htmlFor="start">Start Date</label>
        <input
          id="start"
          type="date"
          value={startDate}
          onChange={handleUpdateStartDate}
        />
      </div>
      <div className={classes.control}>
        <label htmlFor="end">End Date</label>
        <input
          id="end"
          type="date"
          value={endDate}
          onChange={handleUpdateEndDate}
        />
      </div>
      <p className={classes.difference}>
        Difference: {daysDiff} days
      </p>
    </div>
  );
}

export default DateCalculator;
```

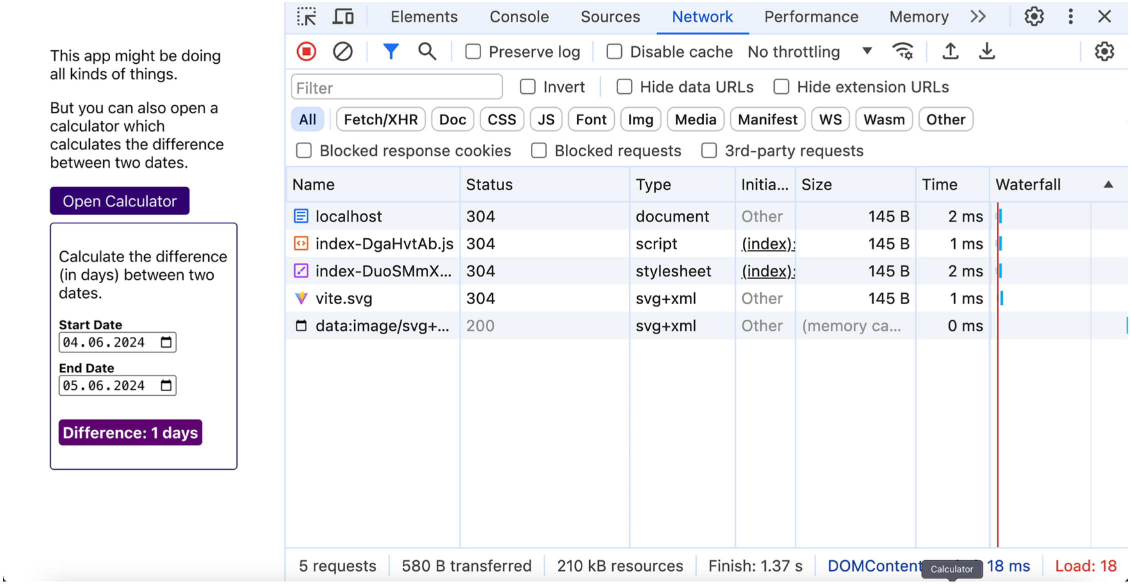
This `DateCalculator` component is then rendered conditionally by the `App` component:

```
import { useState } from 'react';

import DateCalculator from './components/DateCalculator.jsx';

function App() {
  const [showDateCalc, setShowDateCalc] = useState(false);

  function handleOpenDateCalc() {
    setShowDateCalc(true);
  }

  return (
    <>
      <p>This app might be doing all kinds of things.</p>
      <p>
        But you can also open a calculator which calculates
        the difference between two dates.
      </p>
      <button onClick={handleOpenDateCalc}>Open Calculator</button>
      {showDateCalc && <DateCalculator />}
    </>
  );
}

export default App;
```

In this example, the `DateCalculator` component uses a third-party library (the `date-fns` library) to access various date-related utility functions (for example, a function for calculating the difference between two dates, or `differenceInDays`).

The component then accepts two date values and calculates the difference between those dates in days—though the actual logic of the component isn't too important here. What is important is the fact that a third-party library and various utility functions are used. This adds quite a bit of JavaScript code to the overall code bundle, and all that code must be downloaded when the entire website is loaded for the first time, even though the date calculator isn't even visible at that point in time (because it is rendered conditionally).

After building the app for production (via `npm run build`), when previewing that production version (via `npm run preview`), you can see one main code bundle file being downloaded in the following screenshot:


Elements
Console
Sources
Network
Performance
Memory
☐ Preserve log
☐ Disable cache
No throttling
Filter
☐ Invert
☐ Hide data URLs
☐ Hide extension URLs
All
Fetch/XHR
Doc
CSS
JS
Font
Img
Media
Manifest
WS
Wasm
Other
☐ Blocked response cookies
☐ Blocked requests
☐ 3rd-party requests
Open Calculator
Calculator

This app might be doing all kinds of things.

But you can also open a calculator which calculates the difference between two dates.

Calculate the difference (in days) between two dates.

Start Date 04.06.2024

End Date 05.06.2024

Difference: 1 days

Name	Status	Type	Initia...	Size	Time	Waterfall
localhost	304	document	Other	145 B	2 ms	
index-DgaHvtAb.js	304	script	(index):	145 B	1 ms	
index-DuoSMmX...	304	stylesheet	(index):	145 B	2 ms	
vite.svg	304	svg+xml	Other	145 B	1 ms	
data:image/svg+...	200	svg+xml	Other	(memory ca...	0 ms	

5 requests | 580 B transferred | 210 kB resources | Finish: 1.37 s | DOMContent 18 ms | Load: 18

Figure 10.10: One main bundle file is downloaded

The `Network` tab in the browser's developer tools reveals outgoing network requests. As you can see in the screenshot, one main JavaScript bundle file is downloaded. You won't see any extra requests being sent when the button is clicked. This implies that all the code, including the code needed for `DateCalculator`, was downloaded upfront.

That's where code splitting with React's `lazy()` function becomes useful.

This function can be wrapped around a dynamic import to load the imported component only once it's needed.

Note

Dynamic imports are a native JavaScript feature that allows for dynamically importing JavaScript code files. For further information on this topic, visit `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/import`.

In the preceding example, it would be used like this in the `App` component file:

```
import { lazy, useState } from 'react';

const DateCalculator = lazy(() => import(
    './components/DateCalculator.jsx'
  )
);

function App() {
  const [showDateCalc, setShowDateCalc] = useState(false);

  function handleOpenDateCalc() {
    setShowDateCalc(true);
  }

  return (
    <>
      <p>This app might be doing all kinds of things.</p>
      <p>
        But you can also open a calculator which calculates
        the difference between two dates.
      </p>
      <button onClick={handleOpenDateCalc}>Open Calculator</button>
      {showDateCalc && <DateCalculator />}
    </>
  );
}

export default App;
```

This alone won't do the trick though. You must also wrap the conditional JSX code, where the dynamically imported component is used, with another component provided by React – the `<Suspense>` component – like this:

```
import { lazy, Suspense, useState } from 'react';

const DateCalculator = lazy(() => import(
    './components/DateCalculator.jsx'
  )
);

function App() {
  const [showDateCalc, setShowDateCalc] = useState(false);

  function handleOpenDateCalc() {
    setShowDateCalc(true);
  }

  return (
    <>
      <p>This app might be doing all kinds of things.</p>
      <p>
        But you can also open a calculator which calculates
        the difference between two dates.
      </p>
      <button onClick={handleOpenDateCalc}>Open Calculator</button>
      <Suspense fallback={<p>Loading...</p>}>
        {showDateCalc && <DateCalculator />}
      </Suspense>
    </>
  );
}

export default App;
```

Note

You can find the finished example code on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/10-behind-scenes/examples/06-code-splitting`.

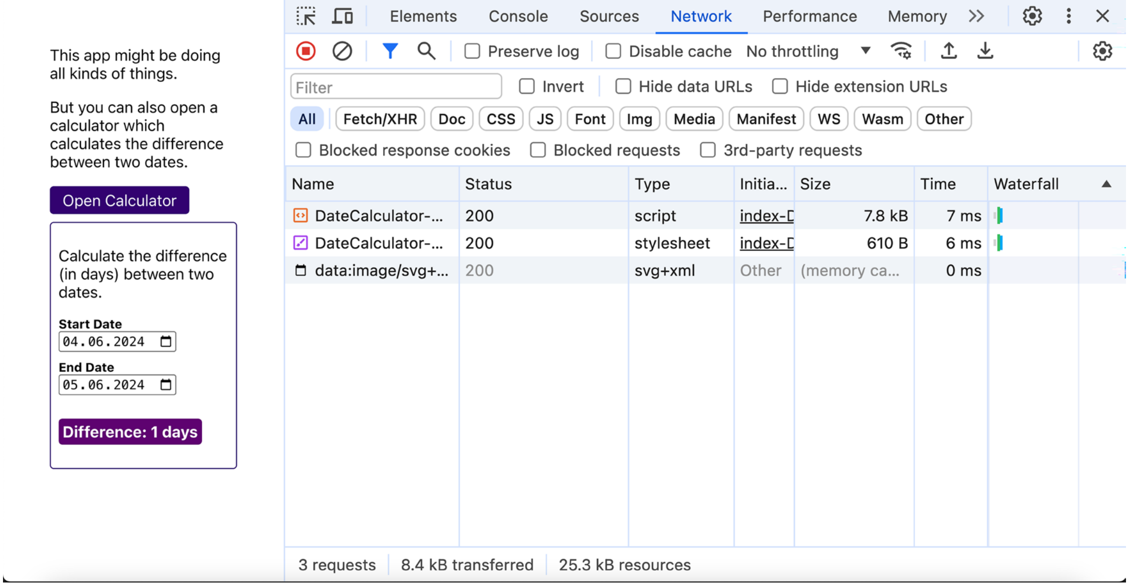
`Suspense` is a component built into React that aims to display fallback content while some resource or data is loading. Therefore, when using it for lazy loading, you must wrap it around any conditional code that uses React's `lazy()` function. `Suspense` also has one mandatory prop that must be provided, the `fallback` prop, which expects a JSX value that will be rendered as fallback content until the dynamically loaded content is available.

`lazy()` leads to the overall JavaScript code being split up into multiple bundles. The bundle that contains the `DateCalculator` component (and its dependencies, such as the `date-fns` library code) is only downloaded when it's needed—that is, when the button in the `App` component is clicked. If that download were to take a bit longer, the `fallback` content of `Suspense` would be shown on the screen in the meantime.

Note

React's `Suspense` component is not limited to being used in conjunction with the `lazy()` function. *Chapter 14, Managing Data with React Router*, and *Chapter 17, Understanding React Suspense & The use() Hook*, will explore how the `Suspense` component may be used to show fallback content while fetching data.

After adding `lazy()` and the `Suspense` component as described, a smaller bundle is initially downloaded. In addition, if the button is clicked, more code files are downloaded:



This app might be doing all kinds of things.

But you can also open a calculator which calculates the difference between two dates.

Calculate the difference (in days) between two dates.

Start Date: 04.06.2024

End Date: 05.06.2024

Difference: 1 days

Name	Status	Type	Initia...	Size	Time
DateCalculator-...	200	script	index-D	7.8 kB	7 ms
DateCalculator-...	200	stylesheet	index-D	610 B	6 ms
data:image/svg+...	200	svg+xml	Other	(memory ca...	0 ms

3 requests | 8.4 kB transferred | 25.3 kB resources

Open Calculator
Elements
Console
Sources
Network
Performance
Memory
☐ Preserve log
☐ Disable cache
No throttling
Filter
☐ Invert
☐ Hide data URLs
☐ Hide extension URLs
All
Fetch/XHR
Doc
CSS
JS
Font
Img
Media
Manifest
WS
Wasm
Other
☐ Blocked response cookies
☐ Blocked requests
☐ 3rd-party requests
Waterfall

Figure 10.11: After clicking the button, an extra code file is downloaded

Just as with all the other optimization techniques described thus far, the `lazy()` function is not a function you should start wrapping around all your imports. If an imported component is very small and simple (and doesn't use any third-party code), splitting the code isn't really worth it, especially since you have to consider that the additional HTTP request required for downloading the extra bundle also comes with some overhead.

It also doesn't make sense to use `lazy()` on components that will be loaded initially anyway. Only consider using it on conditionally loaded components.

Strict Mode

Throughout this chapter, you have learned a lot about React's internals and various optimization techniques. Not really an optimization technique, but still related, is another feature offered by React, called **Strict Mode**.

You may have stumbled across code like this before:

```
import React from 'react';

// ... other code ...
root.render(<React.StrictMode><App /></React.StrictMode >);
```

`<React.StrictMode>` is another built-in component provided by React. It doesn't render a visual element, but it will enable some extra checks that are performed behind the scenes by React.

Most checks are related to identifying the use of unsafe or legacy code (i.e., features that will be removed in the future). But there are also some checks that aim to help you identify potential problems with your code.

For example, when using Strict Mode, React will execute component functions twice and also unmount and remount every component whenever it mounts for the first time. This is done to ensure that you're managing your state and side effects in a consistent and correct way (for example, that you do have cleanup functions in your effect functions).

Note

Strict Mode only affects your app and its behavior during development. It does not influence your app once you build it for production. Extra checks of effects such as double component function execution will not be performed in production.

Building React apps with Strict Mode enabled can sometimes lead to confusion or annoying error messages. You might, for example, wonder why your component effects are executing too often.

Therefore, it's your personal decision whether you want to use Strict Mode or not. Enabling it can help you catch and fix errors early though.

Debugging Code and the React Developer Tools

Earlier in this chapter, you learned that component functions may execute quite frequently and that you can prevent unnecessary executions using `memo()` and `useMemo()` (and that you shouldn't always prevent them).

Identifying component executions by adding `console.log()` inside the component functions is one way of gaining insight into a component. It's the approach used throughout this chapter. However, for large React apps with dozens, hundreds, or even thousands of components, using `console.log()` can get tedious.

That's why the React team also built an official tool to help with gaining app insights. React Developer Tools is an extension that can be installed on all major browsers (Chrome, Firefox, and Edge). You can find and install the extension by simply searching the web for `"<your browser> react developer tools"` (e.g., *chrome react developer tools*).

Once you have installed the extension, you can access it directly from inside the browser. For example, when using Chrome, you can access the React Developer Tools extension directly from inside Chrome's developer tools (which can be opened via the menu in Chrome). Explore the specific extension documentation (in your browser's extensions store) for details on how to access it.

The React Developer Tools extension offers two areas: a `Components` page and a `Profile` page:

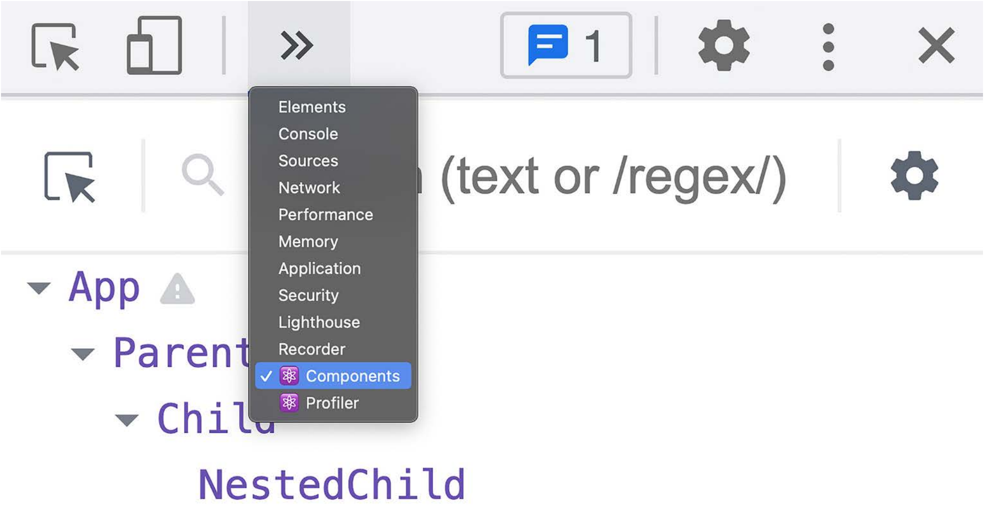
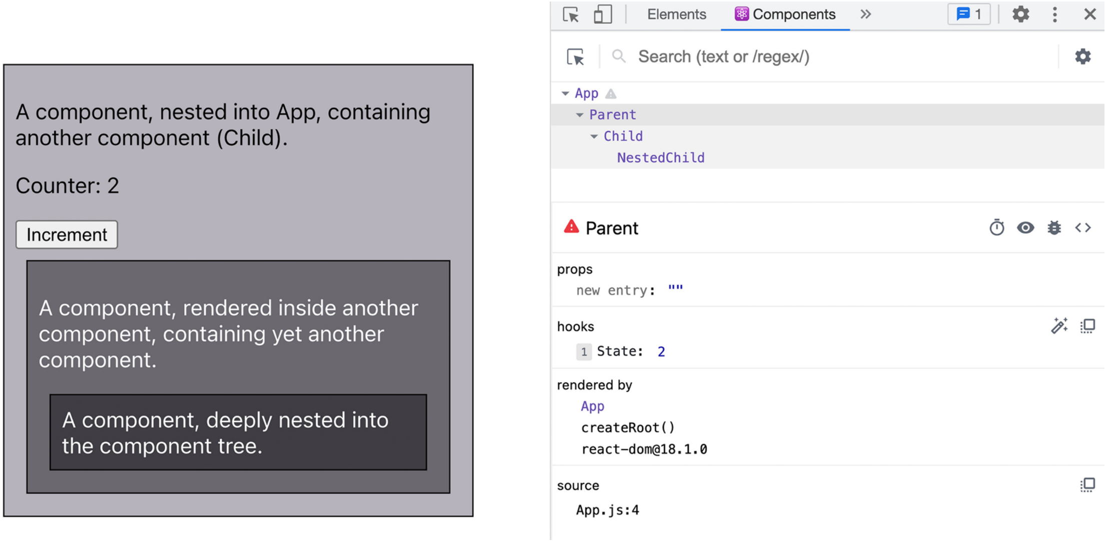

1
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
Recorder
✓ Components
Profiler
(text or /regex/)
App
Parent
Child
NestedChild

Figure 10.12: React Developer Tools can be accessed via browser developer tools

The `Components` page can be used to analyze the component structure of the currently rendered page. You can use this page to understand the structure of your components (i.e., the "tree of components"), how components are nested into each other, and even the configuration (props, state) of components.



A component, nested into App, containing another component (Child).

Counter: 2

A component, rendered inside another component, containing yet another component.

A component, deeply nested into the component tree.

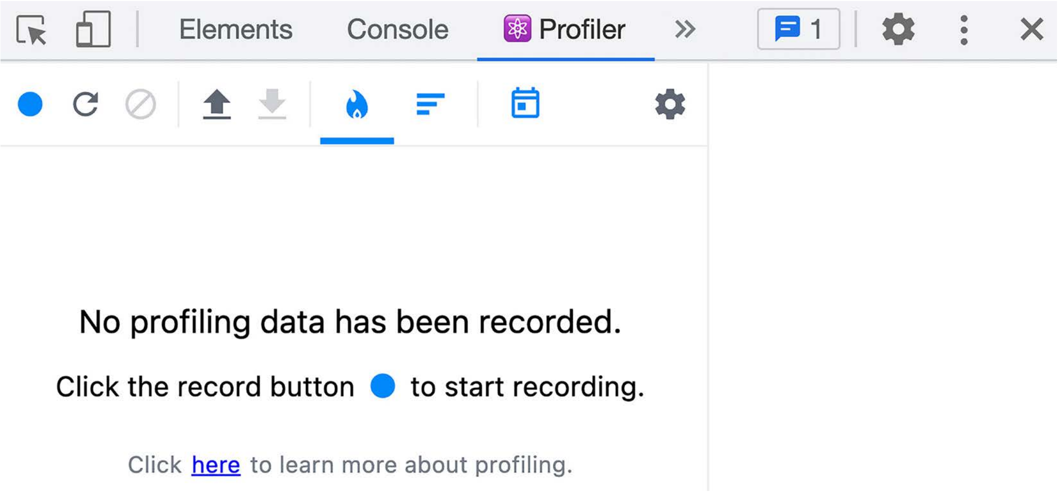
Increment
Elements
Components
1
Search (text or /regex/)
App
Parent
Child
NestedChild
Parent

props
new entry: ""
hooks
1 State: 2
rendered by
App
createRoot()
react-dom@18.1.0
source
App.js:4

Figure 10.13: Component relations and data are shown

This page can be very useful when attempting to understand the current state of a component, how a component is related to other components, and which other components may therefore influence a component (e.g., cause it to be re-evaluated).

However, in the context of this chapter, the more useful page is the `Profiler` page:



No profiling data has been recorded.

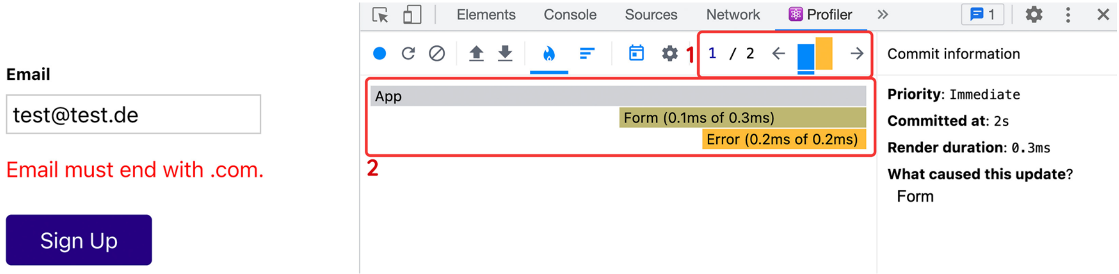
Click the record button ● to start recording.

Click here to learn more about profiling.

Figure 10.14: The Profiler page (without any data gathered)

On this page, you can begin recording component evaluations (i.e., component function executions). You can do this by simply clicking the `Record` button in the top-left corner (the blue circle). This button will then be replaced by a `Stop` button, which you can click to end the recording.

After recording the React app for a couple of seconds (and interacting with it during that period), an example result could look like this:



Email

test@test.de

Email must end with .com.

App

Form (0.1ms of 0.3ms)

Error (0.2ms of 0.2ms)

Commit information

Priority: `Immediate`

Committed at: `2s`

Render duration: `0.3ms`

What caused this update?

Form

Figure 10.15: The Profiler page shows various bars after recording has finished

This result consists of two main areas:

- A list of bars, indicating the number of component re-evaluations (every bar reflects one re-evaluation cycle that affected one or more components). You can click these bars to explore more details about a specific cycle.

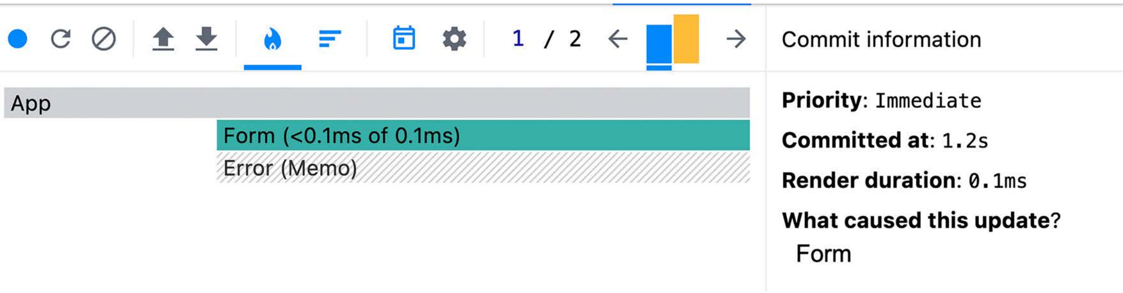
- For the selected evaluation cycle, a list of the affected components is presented. You can identify affected components easily as their bars are colored and timing information is displayed for them.

You can select any render cycle from *1* (in this case, there are two for this recording session) to view which components were affected. The bottom part of the window (*2*) shows all affected components by highlighting them with some color and outputting the overall amount of time taken by the components to be re-evaluated (for example, `0.1ms of 0.3ms`).

> **Note**
>
> It's worth noting that this tool also proves that component evaluation is extremely fast—`0.1ms` for re-evaluating a component is way too fast for any human to realize that something happened behind the scenes.

On the right side of the window, you also learn more about this component evaluation cycle. For example, you learn where it was triggered. In this case, it was triggered by the `Form` component (it's the same example as discussed earlier in this chapter, in the *Avoiding Unnecessary Child Component Evaluations* section).

The `Profiler` page can therefore also help you to identify component evaluation cycles and determine which components are affected. In this example, you can see a difference if the `memo()` function is wrapped around the `Error` component:



Commit information

Priority: `Immediate`

Committed at: `1.2s`

Render duration: `0.1ms`

What caused this update?
Form

Figure 10.16: Only the Form component is affected, not the Error component

After re-adding the `memo()` function as a wrapper around the `Error` component (as explained earlier in this chapter), you can use the `Profiler` page of React Developer Tools to confirm that the `Error` component is no longer unnecessarily evaluated. To do this, you should start a new recording session and reproduce the situation, where previously, without `memo()`, the `Error` component would've been called again.

The diagonal grayed-out lines across the `Error` component in the `Profiler` window signal that this component was not affected by some other component function invocation.

React Developer Tools can therefore be used to gain deeper insights into your React app and your components. You can use them in addition or instead of calling `console.log()` in a component function.

Summary and Key Takeaways

- React components are re-evaluated (executed) whenever their state changes or the parent component is evaluated.
- React optimizes component evaluation by calculating required UI changes with the help of a virtual DOM first.
- Multiple state updates that occur at the same time and in the same place are batched together by React. This ensures that unnecessary component evaluations are avoided.
- The `memo()` function can be used to control component function executions.
- `memo()` looks for prop value differences (old props versus new props) to determine whether a component function must be executed again.
- `useMemo()` can be used to wrap performance-intensive computations and only perform them if key dependencies changed.
- Both `memo()` and `useMemo()` should be used carefully since they also come at a cost (the comparisons performed).
- When working with React 19 or higher, you can install and enable the (experimental) React compiler to automatically optimize your code during the build process.
- The initial code download size can be reduced with the help of code splitting via the `lazy()` function (in conjunction with the built-in `Suspense` component)
- React's Strict Mode can be enabled (via the built-in `<React.StrictMode>` component) to perform various extra checks and detect potential bugs in your application.
- React Developer Tools can be used to gain deeper insights into your React app (for example, component structure and re-evaluation cycles).

What's Next?

As a developer, you should always know and understand the tool you're working with—in this case, React.

This chapter allowed you to get a better idea of how React works under the hood and which optimizations are implemented automatically. In addition, you also learned about various optimization techniques that can be implemented by you.

The next chapter will go back to solving actual problems you might face when trying to build React apps. Instead of optimizing React apps, you will learn more about techniques and features that can be used to solve more complex problems related to component and application state management.

Test Your Knowledge!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to examples that can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/10-behind-scenes/exercises/questions-answers.md`:

1. Why does React use a virtual DOM to detect required DOM updates?
2. How is the real DOM affected when a component function is executed?
3. Which components are great candidates for the `memo()` function? Which components are bad candidates?
4. How is `useMemo()` different from `memo()`?
5. What's the idea behind code splitting and the `lazy()` function?

Apply What You Learned

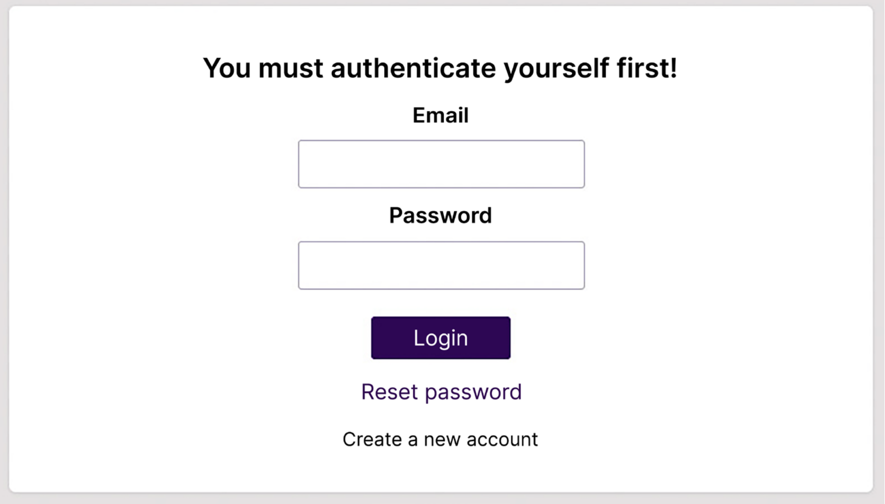
With your newly gained knowledge about React's internals and some of the optimization techniques you can employ in order to improve your apps, you can now apply this knowledge in the following activity.

Activity 10.1: Optimize an Existing App

In this activity, you're handed an existing React app that could be optimized in various places. Your task is to identify optimization opportunities and implement appropriate solutions. Keep in mind that too much optimization can actually lead to a worse result.

> **Note**
>
> You can find the starting code for this activity at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/10-behind-scenes/activities/practice-1-start`. When downloading this code, you'll always download the entire repository. Make sure to then navigate to the subfolder with the starting code (`activities/practice-1-start` in this case) to use the right code snapshot.
>
> The provided project also uses many features covered in earlier chapters. Take the time to analyze it and understand the provided code. This is a great practice and allows you to see many key concepts in action.

Once you have downloaded the code and run `npm install` in the project folder (to install all required dependencies), you can start the development server via `npm run dev`. As a result, upon visiting `localhost:5173`, you should see the following UI:



You must authenticate yourself first!

Email

Password

Login
Reset password
Create a new account

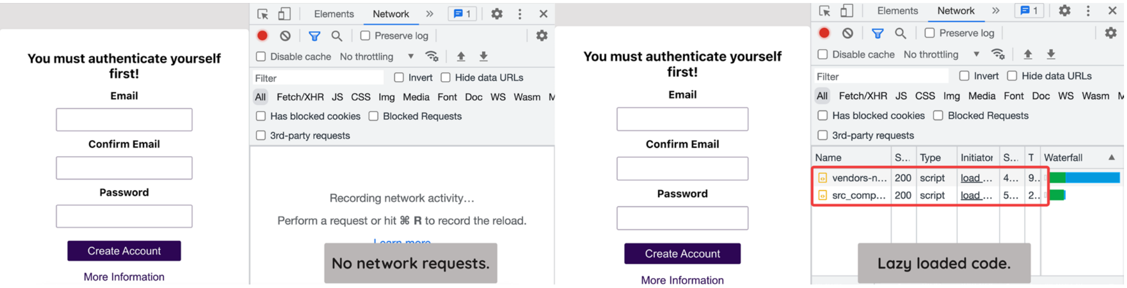
Figure 10.17: The running starting project

Take your time to get acquainted with the provided project. Experiment with the different buttons in the UI, fill in some dummy data in the form input fields, and analyze the provided code. Please note that this dummy project does not send any HTTP requests to any server. All entered data is discarded the moment it is entered.

To complete the activity, the solution steps are as follows:

1. Find optimization opportunities by looking for unnecessary component function executions.
2. Also, identify unnecessary code execution inside of component functions (where the overall component function invocation can't be prevented).
3. Determine which code could be loaded lazily instead of eagerly.
4. Use the `memo()` function, the `useMemo()` Hook, and React's `lazy()` function to improve the code.

You can tell that you came up with a good solution and sensible adjustments if you can see extra code fetching network requests (in the **Network** tab of your browser developer tools) for clicking on the **Reset password** or **Create a new account** buttons:



You must authenticate yourself first!

Email

Confirm Email

Password

Recording network activity…

Perform a request or hit ⌘ R to record the reload.

No network requests.

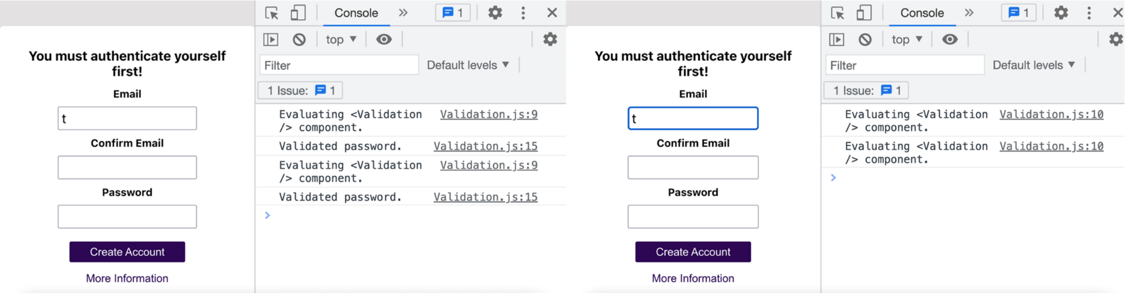
Name	S…	Type	Initiator	S…	T
vendors-n…	200	script	load …	4…	9..
src_comp…	200	script	load …	5…	2..

Lazy loaded code.

Create Account
More Information
Elements
Network
Preserve log
Disable cache
No throttling
Filter
Invert
Hide data URLs
All Fetch/XHR JS CSS Img Media Font Doc WS Wasm
Has blocked cookies
Blocked Requests
3rd-party requests
Learn more
Waterfall

Figure 10.18: In the final solution, some code is lazy loaded

In addition, you should see no `Validated password.` console message when typing into the email input fields (**Email** and **Confirm Email**) of the signup form (that is, the form you switch to when clicking **Create a new account**):

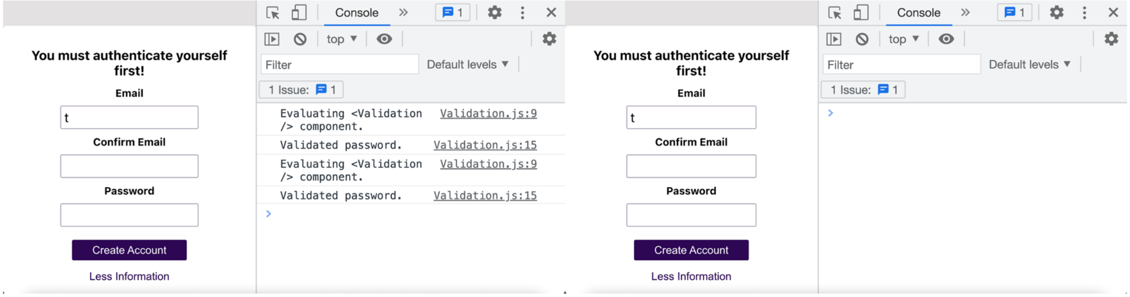

You must authenticate yourself first!
Email
t
Confirm Email
Password
Create Account
More Information
Console
top
Filter
Default levels
1 Issue: 1

```
Evaluating <Validation    Validation.js:9
/> component.
Validated password.      Validation.js:15
Evaluating <Validation    Validation.js:9
/> component.
Validated password.      Validation.js:15
```

```
Evaluating <Validation    Validation.js:10
/> component.
Evaluating <Validation    Validation.js:10
/> component.
```

Figure 10.19: No "Validated password." output in the console

You also shouldn't get any console outputs when clicking the **More Information** button:


You must authenticate yourself first!
Email
t
Confirm Email
Password
Create Account
Less Information
Console
top
Filter
Default levels
1 Issue: 1

```
Evaluating <Validation    Validation.js:9
/> component.
Validated password.      Validation.js:15
Evaluating <Validation    Validation.js:9
/> component.
Validated password.      Validation.js:15
```

Figure 10.20: No console messages when clicking "More Information"

Note

All code files used for this activity, and the solution, can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/10-behind-scenes/activities/practice-1`.

11

Working with Complex State

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Manage cross-component or even app-wide state (instead of just component-specific state)
- Distribute data across multiple components
- Handle complex state values and changes

Introduction

State is one of the core concepts you must understand (and work with) to use React effectively. Basically, every React app utilizes (many) state values across many components to present a dynamic, reactive user interface.

From simple state values that contain a changing counter or values entered by users, all the way up to more complex state values such as the combination of multiple form inputs or user authentication information, state is everywhere. In React apps, it's typically managed with the help of the `useState()` Hook.

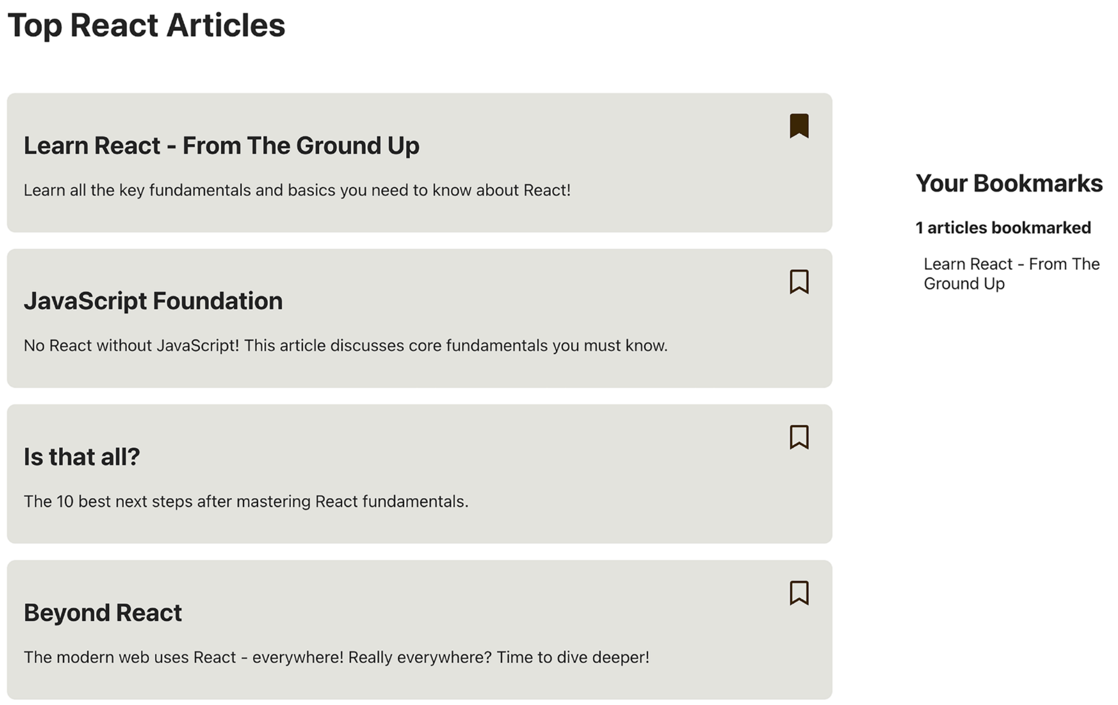
However, once you start building more complex React applications (e.g., online shops, admin dashboards, and similar sites), it is likely that you'll face various challenges related to state. State values might be used in component A but changed in component B or be made up of multiple dynamic values that may change for a broad variety of reasons (e.g., a cart in an online shop, which is a combination of products, where every product has a quantity, a price, and possibly other traits that may be changed individually).

You can handle all these problems with `useState()`, props, and the other concepts covered by this book thus far. But you will notice that solutions based on `useState()` alone gain a complexity that can be difficult to understand and maintain. That's why React has more tools to offer—tools created for these kinds of problems, which this chapter will highlight and discuss.

A Problem with Cross-Component State

You don't even need to build a highly sophisticated React app to encounter a common problem: state that spans multiple components.

For example, you might be building a news app where users can bookmark certain articles. A simple user interface could look like this:



Top React Articles

Learn React - From The Ground Up

Learn all the key fundamentals and basics you need to know about React!

JavaScript Foundation

No React without JavaScript! This article discusses core fundamentals you must know.

Is that all?

The 10 best next steps after mastering React fundamentals.

Beyond React

The modern web uses React - everywhere! Really everywhere? Time to dive deeper!

Your Bookmarks

1 articles bookmarked

Learn React - From The Ground Up

Figure 11.1: An example user interface

As you can see in the preceding figure, the list of articles is on the left, and a summary of the bookmarked articles can be found in the sidebar on the right.

A common solution is to split this user interface into multiple components. The list of articles, specifically, would probably be in its own component—just like the bookmark summary sidebar.

However, in that scenario, both components would need to access the same shared state—that is, the list of bookmarked articles. The article list component would require access in order to add (or remove) articles. The bookmark summary sidebar component would require it as it needs to display the bookmarked articles.

The component tree and state usage for this kind of app could look like this:

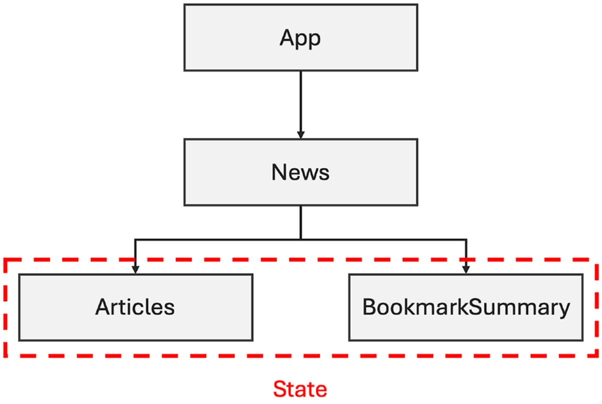

App
News
Articles
BookmarkSummary
State

Figure 11.2: Two sibling components share the same state

In this figure, you can see that the state is shared across these two components. You also see that the two components have a shared parent component (the `News` component, in this example).

Since the state is used by two components, you would not manage it in either of those components. Instead, it's *lifted up*, as described in *Chapter 4*, *Working with Events and State* (in the *Lifting State Up* section). When lifting state up, the state values and pointers to the functions that manipulate the state values are passed down to the actual components that need access via props.

This works and is a common pattern. You can (and should) keep on using it. But what if a component that needs access to some shared state is deeply nested in other components? What if the app component tree from the preceding example looked like this?

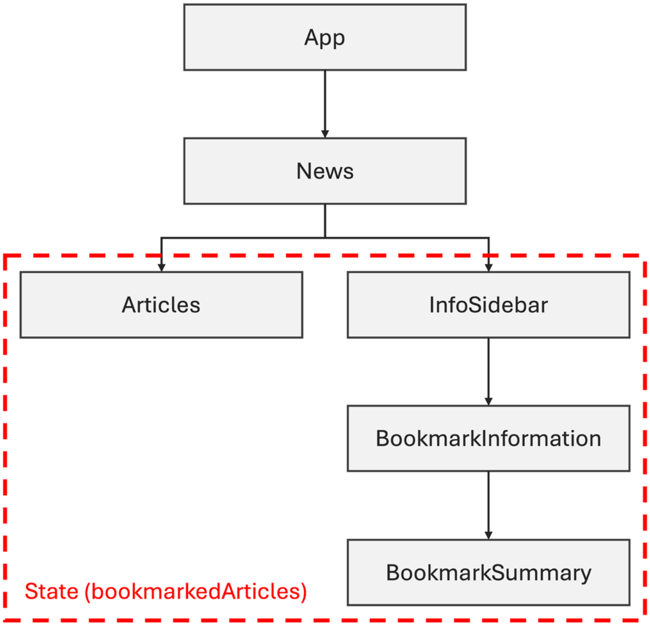

App
News
Articles
InfoSidebar
BookmarkInformation
BookmarkSummary
State (bookmarkedArticles)

Figure 11.3: A component tree with multiple layers of state-dependent components

In this figure, you can see that the `BookmarkSummary` component is a deeply nested component. Between it and the `News` component (which manages the shared state), you have two other components: the `InfoSidebar` component and the `BookmarkInformation` component. In more complex React apps, having multiple levels of component nesting, as in this example, is very common.

Of course, even with those extra components, state values can still be passed down via props. You just need to add props to **all** components between the component that holds the state and the component that needs the state. For example, you must pass the `bookmarkedArticles` state value to the `InfoSidebar` component (via props) so that that component can forward it to `BookmarkInformation`:

```
import BookmarkInformation from
  '../BookmarkSummary/BookmarkInformation.jsx';
import classes from './InfoSidebar.module.css';

function InfoSidebar({ bookmarkedArticles }) {
  return (
    <aside className={classes.sidebar}>
      <BookmarkInformation bookmarkedArticles={bookmarkedArticles} />
    </aside>
  );
}

export default InfoSidebar;
```

The same procedure is repeated inside of the `BookmarkInformation` component.

Note

You can find the complete example on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/11-complex-state/examples/01-cross-cmp-state`.

This kind of pattern is called **prop drilling**. Prop drilling means that a state value is passed through multiple components via props. And it's passed through components that don't need the state themselves at all—except for forwarding it to a child component (as the `InfoSidebar` and `BookmarkInformation` components are doing in the preceding example).

As a developer, you will typically want to avoid this pattern because prop drilling has a few weaknesses:

- Components that are part of prop drilling (such as `InfoSidebar` or `BookmarkInformation`) are not really reusable anymore because any component that wants to use them has to provide a value for the forwarded state prop.
- Prop drilling also leads to a lot of overhead code that has to be written (the code to accept props and forward props).

- Refactoring components becomes more work because state props have to be added or removed.

For these reasons, prop drilling is only acceptable if all components involved are only used in this specific part of the overall React app, and the probability of reusing or refactoring them is low.

Since prop drilling should be avoided (in most situations), React offers an alternative: the **context** API.

Using Context to Handle Multi-Component State

React's context feature is one that allows you to create a value that can easily be shared across as many components as needed, without using props.

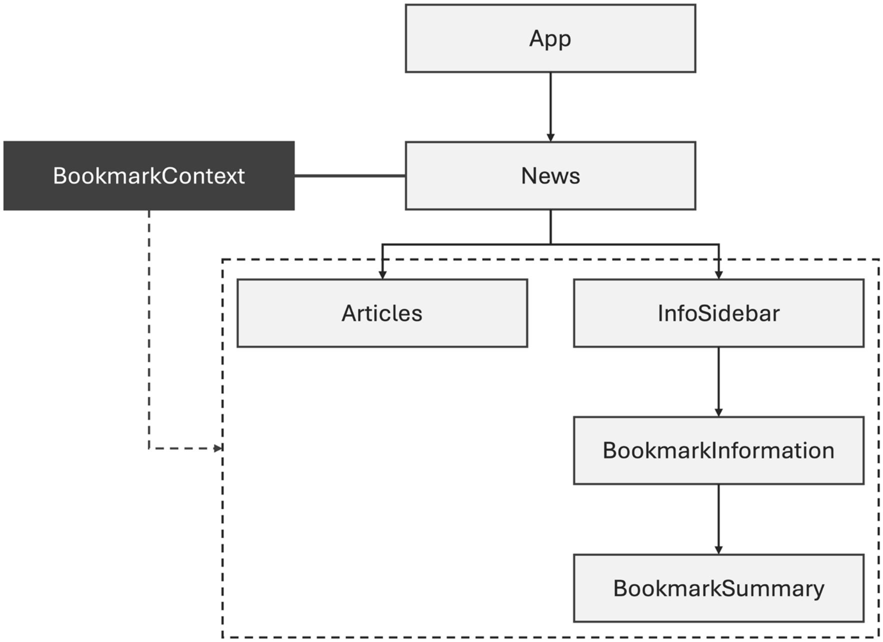


Figure 11.4: React Context is attached to components to expose it to all child components—without prop drilling

Using the context API is a multi-step process, the steps for which are described here:

1. You must create a context value that should be shared.
2. The context must be provided in a parent component of the components that need access to the context object.
3. Components that need access (for reading or writing) must subscribe to the context.

React manages the context value (and its changes) internally and automatically distributes it to all components that have subscribed to the context.

Before any component may subscribe, however, the first step is to create a context object. This is done via React's `createContext()` function:

```
import { createContext } from 'react';
```

```
createContext('Hello Context'); // a context with an initial string value
createContext({}); // a context with an initial (empty) object as a value
```

This function takes an initial value that should be shared. It can be any kind of value (e.g., a string or a number), but typically, it's an object. This is because most shared values are a combination of the actual values and functions that should manipulate those values. All these things are then grouped together into a single context object.

Of course, the initial context value can also be an empty value (e.g., `null`, `undefined`, an empty `string`, etc.) if needed.

`createContext()` also returns a value: a context object that should be stored in a capitalized variable (or constant) because it's actually a React component (and React components should start with capital characters).

Here's how the `createContext()` function can be called to create a context object for the example discussed earlier in this chapter:

```
import { createContext } from 'react';

const BookmarkContext = createContext({
  bookmarkedArticles: []
});

export default BookmarkContext;
```

Here, the initial value is an object that contains the `bookmarkedArticles` property, which holds an (empty) array. You could also store just the array as an initial value (i.e., `createContext([])`) but an object is better since more will be added to it later in the chapter.

This code is typically placed in a separate context code file (e.g., `bookmark-context.jsx`) that's often stored in a folder named `store` (because this context feature can be used as a central state store) or `context`. However, this is just a convention and is not technically required. You can put this code anywhere in your React app.

If the file only contains the above code, it may use `.js` as a file extension since it doesn't contain any JSX code. Later in this chapter, this will change—therefore you can already use `.jsx` as an extension.

Of course, this initial value is not a replacement for state; it's a static value that never changes. But this was just the first of three steps related to context. The next step is to provide the context.

Providing and Managing Context Values

In order to use context values in other components, you must first provide the value. This is done using the value returned by `createContext()`.

When using React 19 or higher, that function yields a React component that should be wrapped around all other components that need access to the context value.

When using an older version of React (i.e., React 18 or older), the value returned by `createContext()` instead is an object that contains a nested `Provider` property. That property then holds a React component that should be wrapped around all other components that need access to the context value.

So, either way, it's all about wrapping components with a context provider component.

In the preceding example, using React 19 or higher, the `BookmarkContext` component returned by `createContext()` could be used in the `News` component to wrap it around both the `Articles` and `InfoSidebar` components:

```
import Articles from '../Articles/Articles.jsx';
import InfoSidebar from '../InfoSidebar/InfoSidebar.jsx';
import BookmarkContext from '../../store/bookmark-context.jsx';

function News() {
  return (
    <BookmarkContext>
      <Articles />
      <InfoSidebar />
    </BookmarkContext>
  );
}
```

Or, if using React 18 or lower:

```
import Articles from '../Articles/Articles.jsx';
import InfoSidebar from '../InfoSidebar/InfoSidebar.jsx';
import BookmarkContext from '../../store/bookmark-context.jsx';

function News() {
  return (
    <BookmarkContext.Provider>
      <Articles />
      <InfoSidebar />
    </BookmarkContext.Provider>
  );
}
```

However, this code does not work because one important thing is missing: the component expects a `value` prop, which should contain the current context value that should be distributed to interested components. While you do provide an initial context value (which could have been empty), you also need to inform React about the current context value because, very often, context values change (they are often used as a replacement for the cross-component state, after all).

Hence, the code could be altered like this when using React 19 or higher:

```
import Articles from '../Articles/Articles.jsx';
import InfoSidebar from '../InfoSidebar/InfoSidebar.jsx';
import BookmarkContext from '../../store/bookmark-context.jsx';

function News() {
  const bookmarkCtxValue = {
    bookmarkedArticles: []
  }; // for now, it's the same value as used before, for the initial context

  return (
    <BookmarkContext value={bookmarkCtxValue}>
      <Articles />
      <InfoSidebar />
    </BookmarkContext>
  );
}
```

Or, if using React 18 or lower:

```
import Articles from '../Articles/Articles.jsx';
import InfoSidebar from '../InfoSidebar/InfoSidebar.jsx';
import BookmarkContext from '../../store/bookmark-context.jsx';

function News() {
  const bookmarkCtxValue = {
    bookmarkedArticles: []
  }; // for now, it's the same value as used before, for the initial context

  return (
    <BookmarkContext.Provider value={bookmarkCtxValue}>
      <Articles />
      <InfoSidebar />
    </BookmarkContext.Provider>
  );
}
```

With this code, an object with a list of bookmarked articles is distributed to interested descendent components.

The list is still static, though. But that can be changed with a tool you already know about: the `useState()` Hook. Inside the `News` component, you can use the `useState()` Hook to manage the list of bookmarked articles, like this:

```
import { useState } from 'react';

import Articles from '../Articles/Articles.jsx';
import InfoSidebar from '../InfoSidebar/InfoSidebar.jsx';
import BookmarkContext from '../../store/bookmark-context.jsx';

function News() {
  const [savedArticles, setSavedArticles] = useState([]);

  const bookmarkCtxValue = {
    bookmarkedArticles: savedArticles // using the state as a value
  };

  return (
    <BookmarkContext value={bookmarkCtxValue}>
      <Articles />
      <InfoSidebar />
    </BookmarkContext>
  );
}
```

With this change, the context changes from static to dynamic. Whenever the `savedArticles` state changes, the context value will change.

Therefore, that's the missing piece when it comes to providing the context. If the context should be dynamic (and changeable from inside some nested child component), the context value should also include a pointer to the function that triggers a state update.

For the preceding example, the code is therefore adjusted like this:

```
import { useState } from 'react';

import Articles from '../Articles/Articles.jsx';
import InfoSidebar from '../InfoSidebar/InfoSidebar.jsx';
import BookmarkContext from '../../store/bookmark-context.jsx';

function News() {
  const [savedArticles, setSavedArticles] = useState([]);
  function addArticle(article) {
    setSavedArticles(
     (prevSavedArticles) => [...prevSavedArticles, article]
    );
  }

  function removeArticle(articleId) {
    setSavedArticles(
      (prevSavedArticles) => prevSavedArticles
        .filter(
          (article) => article.id !== articleId
        )
    );
  }

  const bookmarkCtxValue = {
    bookmarkedArticles: savedArticles,
    bookmarkArticle: addArticle,
    unbookmarkArticle: removeArticle
  };

  return (
    <BookmarkContext value={bookmarkCtxValue}>
      <Articles />
      <InfoSidebar />
    </BookmarkContext>
  );
}
```

The following are two important things changed in this code snippet:

- Two new functions were added: `addArticle` and `removeArticle`.
- Properties that point at these functions were added to `bookmarkCtxValue`: the `bookmarkArticle` and `unbookmarkArticle` methods.

The `addArticle` function adds a new article (which should be bookmarked) to the `savedArticles` state. The function form of updating the state value is used since the new state value depends on the previous state value (the bookmarked article is added to the list of already bookmarked articles).

Similarly, the `removeArticle` function removes an article from the `savedArticles` list by filtering the existing list such that all items, except for the one that has a matching `id` value, are kept.

If the `News` component did not use the new context feature, it would be a component that uses state, just as you have seen many times before in this book. But now, by using React's context API, those existing capabilities are combined with a new feature (the context) to create a dynamic, distributable value.

Any components nested in the `Articles` or `InfoSidebar` components (or their descendent components) will be able to access this dynamic context value, and the `bookmarkArticle` and `unbookmarkArticle` methods in the context object, without any prop drilling.

Note

You don't have to create dynamic context values. You could also distribute a static value to nested components. This is possible but a rare scenario, since most React apps typically need dynamic state values that can change across components.

Using Context in Nested Components

With the context created and provided, it's ready to be used by components that need to access or change the context value.

To make the context value accessible to components nested inside the context component (`BookmarkContext`, in the preceding example), React offers a `use()` Hook that can be used.

This Hook, however, is only available when working with React 19 or higher. In projects that use older React versions, you would instead use the `useContext()` Hook for accessing some context value. That Hook is also still supported in React 19, hence you can use either of the two Hooks for getting hold of a context value.

The `use()` Hook is a bit more flexible than the `useContext()` Hook since, unlike any other Hooks, it may actually also be used from inside `if` statements or loops. In addition, the Hook can be used for more than getting access to context values—you'll therefore see `use()` again in *Chapter 17*, *Understanding React Suspense & The use() Hook*.

As mentioned, when working with React 19, if you're trying to get access to a context value, both `use()` and `useContext()` can be used. Both `use()` and `useContext()` require one argument: the context object that was created via `createContext()`, i.e., the value returned by that function. As a result, you'll then get the value passed to the context provider component (i.e., the value set for its `value` prop). When working with React 19 or higher, since `use()` is a bit more flexible and certainly a bit less to type, you can ignore `useContext()` and use the `use()` Hook for accessing context values.

For the preceding example, the context value can be used in the `BookmarkSummary` component like this:

```
import { use } from 'react'; // or import useContext for React < 19

import BookmarkContext from '../../store/bookmark-context.jsx';
import classes from './BookmarkSummary.module.css';

function BookmarkSummary() {
  const bookmarkCtx = use(BookmarkContext);
  // React < 19: const bookmarkCtx = useContext(BookmarkContext);

  const numberOfArticles = bookmarkCtx.bookmarkedArticles.length;

  return (
    <>
      <p className={classes.summary}>
        {numberOfArticles} articles bookmarked
      </p>
      <ul className={classes.list}>
        {bookmarkCtx.bookmarkedArticles.map((article) => (
          <li key={article.id}>{article.title}</li>
        ))}
      </ul>
    </>
  );
}

export default BookmarkSummary;
```

In this code, `use()` receives the `BookmarkContext` value, which is imported from the `store/bookmark-context.jsx` file. It then returns the value stored in the context, which is the `bookmarkCtxValue` found in the previous code example. As you can see in that snippet, `bookmarkCtxValue` is an object with three properties: `bookmarkedArticles`, `bookmarkArticle` (a method), and `unbookmarkArticle` (also a method).

This returned object is stored in a `bookmarkCtx` constant. Whenever the context value changes (because the `setSavedArticles` state-updating function in the `News` component is executed), this `BookmarkSummary` component will also be executed again by React, and thus `bookmarkCtx` will hold the latest state value.

Finally, in the `BookmarkSummary` component, the `bookmarkedArticles` property is accessed on the `bookmarkCtx` object. This list of articles is then used to calculate the number of bookmarked articles, output a short summary, and display the list on the screen.

Similarly, `BookmarkContext` can be used via `use()` in the `Articles` component:

```
import { use } from 'react';
// other imports

function Articles() {
  const bookmarkCtx = use(BookmarkContext);
  // or: const bookmarkCtx = useContext(BookmarkContext)
  return (
    <ul>
      {dummyArticles.map((article) => {
        const isBookmarked = bookmarkCtx.bookmarkedArticles.some(
          (bArticle) => bArticle.id === article.id
        );

        // default icon: Empty bookmark icon, because not bookmarked
        let buttonIcon = <FaRegBookmark />;

        if (isBookmarked) {
          buttonIcon = <FaBookmark />;
        }

        return (
          <li key={article.id}>
            <h2>{article.title}</h2>
            <p>{article.description}</p>
            <button>{buttonIcon}</button>
          </li>
        );
      })}
    </ul>
  );
}
```

In this component, the context is used to determine whether or not a given article is currently bookmarked (this information is required in order to change the icon and functionality of the button).

That's how context values (whether static or dynamic) can be read in components. Of course, they can also be changed, as discussed in the next section.

Changing Context from Nested Components

React's context feature is often used to share data across multiple components without using props. It's therefore also quite common that some components must manipulate that data. For example, the context value for a shopping cart must be adjustable from inside the component that displays product items (because those probably have an `"Add to cart"` button).

However, to change context values from inside a nested component, you cannot simply overwrite the stored context value. The following code would not work as intended:

```
const bookmarkCtx = use(BookmarkContext);
```

```
// Note: This does NOT work
bookmarkCtx.bookmarkedArticles = []; // setting the articles to an empty array
```

This code does not work. Just as you should not try to update state by simply assigning a new value, you can't update context values by assigning a new value. That's why two methods (`bookmarkArticle` and `unbookmarkArticle`) were added to the context value in the *Providing and Managing Context Values* section. These two methods point at functions that trigger state updates (via the state-updating function provided by `useState()`).

Therefore, in the `Articles` component, where articles can be bookmarked or unbookmarked via button clicks, these methods should be called:

```
// This code is part of the Article component function
// default action => bookmark article, because not bookmarked yet
let buttonAction = () => bookmarkCtx.bookmarkArticle(article);
// default button icon: Empty bookmark icon, because not bookmarked
let buttonIcon = <FaRegBookmark />;

if (isBookmarked) {
  buttonAction = () => bookmarkCtx.unbookmarkArticle(article.id);
  buttonIcon = <FaBookmark />;
}
```

The `bookmarkArticle` and `unbookmarkArticle` methods are called inside of anonymous functions that are stored in a `buttonAction` variable. That variable is assigned to the `onClick` prop of the `<button>` (see the previous code snippet).

With this code, the context value can be changed successfully. Thanks to the steps taken in the previous section (*Using Context in Nested Components*), whenever the context value is updated, it is then also automatically reflected in the user interface.

Note

The finished example code can be found on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/11-complex-state/examples/02-cross-cmp-state-with-context`.

Using the Context API Efficiently

Being able to create, provide, access, and change context is important—ultimately, it is these things that allow you to use React's context API in your applications. But as your applications (and therefore probably also your context values) become more complex, it's also important to set up and manage your context efficiently, for example, by getting proper IDE support.

Getting Better Code Completion

In the section *Using Context to Handle Multi-Component State*, a context object was created via `createContext()`. That function received an initial context value—an object that contains a `bookmarkedArticles` property, in the preceding example.

In this example, the initial context value isn't too important. It's not often used because it's overwritten with a new value inside the `News` component regardless. However, depending on which **Integrated Development Environment (IDE)** you're using, you can get better code auto-completion when defining an initial context value that has the same shape and structure as the final context value that will be managed in other React components.

Therefore, since two methods were added to the context value in the section *Providing and Managing Context Values*, those methods should also be added to the initial context value in `store/bookmark-context.jsx`:

```
const BookmarkContext = createContext({
  bookmarkedArticles: [],
  bookmarkArticle: () => {},
  unbookmarkArticle: () => {}
});

export default BookmarkContext;
```

The two methods are added as empty functions that do nothing because the actual logic is set in the `News` component. The methods are only added to this initial context value to provide better IDE auto-completion. This step is therefore optional.

Context or Lifting State Up?

At this point, you now have two tools for managing cross-component state:

- You can lift state up, as described earlier in the book (in *Chapter 4*, *Working with Events and State*, in the *Lifting State Up* section).
- Alternatively, you can use React's context API, as explained in this chapter.

Which of the two approaches should you use in each scenario?

Ultimately, it is up to you how you manage this, but there are some straightforward rules you can follow:

- Lift the state up if you only need to share state across one or two levels of component nesting.
- Use the context API if you have long chains of components (i.e., deep nesting of components) with shared state. Once you start to use a lot of prop drilling, it's time to consider React's context feature.
- Also use the context API if you have a relatively flat component tree but want to reuse components (i.e., you don't want to use props for passing state to components).

Outsourcing Context Logic into Separate Components

With the previously explained steps, you have everything you need to manage cross-component state via context.

But there is one pattern you can consider for managing your dynamic context value and state: creating a separate component for providing (and managing) the context value.

In the preceding example, the `News` component was used to provide the context and manage its (dynamic, state-based) value. While this works, your components can get unnecessarily complex if they have to deal with context management. Creating a separate, dedicated component for that can therefore lead to code that's easier to understand and maintain.

For the preceding example, that means that, inside of the `store/bookmark-context.jsx` file, you could create a `BookmarkContextProvider` component that looks like this:

```
export function BookmarkContextProvider({ children }) {
  const [savedArticles, setSavedArticles] = useState([]);

  function addArticle(article) {
    setSavedArticles(
      (prevSavedArticles) => [...prevSavedArticles, article]
    );
  }

  function removeArticle(articleId) {
    setSavedArticles((prevSavedArticles) =>
      prevSavedArticles.filter((article) => article.id !== articleId)
    );
  }

  const bookmarkCtxValue = {
    bookmarkedArticles: savedArticles,
    bookmarkArticle: addArticle,
    unbookmarkArticle: removeArticle,
  };

  return (
    <BookmarkContext value={bookmarkCtxValue}>
      {children}
    </BookmarkContext>
  );
}
```

This component contains all the logic related to managing a list of bookmarked articles via state. It creates the same context value as before (a value that contains the list of articles as well as two methods for updating that list).

The `BookmarkContextProvider` component does one additional thing though. It uses the special `children` prop (covered in *Chapter 3*, *Components and Props*, in the *The Special "children" Prop* section) to wrap whatever is passed between the `BookmarkContextProvider`'s component tags with `BookmarkContext`.

This allows the use of the `BookmarkContextProvider` component in the `News` component, like so:

```
import Articles from '../Articles/Articles.jsx';
import InfoSidebar from '../InfoSidebar/InfoSidebar.jsx';
import { BookmarkContextProvider } from '../../store/bookmark-context.jsx';

function News() {
  return (
    <BookmarkContextProvider>
      <Articles />
      <InfoSidebar />
    </BookmarkContextProvider>
  );
}

export default News;
```

Instead of managing the entire context value, the `News` component now simply imports the `BookmarkContextProvider` component and wraps that component around `Articles` and `InfoSidebar`. The `News` component, therefore, is leaner.

> **Note**
>
> This pattern is entirely optional. It's neither an official best practice nor does it yield any performance benefits. It's just a pattern that can help with keeping your component functions lean and concise.
>
> It's also worth mentioning that there is a related pattern for consuming context. That pattern, however, relies on building a custom React Hook—a concept that will be covered in the next chapter. Therefore, the mentioned context consumption pattern will be covered in the next chapter, too.

Combining Multiple Contexts

Especially in bigger and more feature-rich React applications, it is possible (and quite probable), that you will need to work with multiple context values that are likely unrelated to each other. For example, an online shop could use one context for managing the shopping cart, another context for the user authentication status, and yet another context value for tracking page analytics.

React fully supports use cases like this. You can create, manage, provide, and use as many context values as needed. You can manage multiple (related or unrelated) values in a single context or use multiple contexts. You can provide multiple contexts in the same component or in different components. It is totally up to you and your app's requirements.

You can also use multiple contexts in the same component (meaning that you can call `use()` or `useContext()` multiple times, with different context values).

Limitations of useState( )

Thus far in this chapter, the complexity of cross-component state has been explored. But state management can also get challenging in scenarios where some state is only used inside a single component.

`useState()` is a great tool for state management in most scenarios (of course, right now, it's also the only tool that's been covered). Therefore, `useState()` should be your default choice for managing state. But `useState()` can reach its limits if you need to derive a new state value that's based on the value of another state variable, as in this example:

```
setIsLoading(fetchedPosts ? false : true);
```

This short snippet is taken from a component where an HTTP request is sent to fetch some blog posts:

```
function App() {
  const [fetchedPosts, setFetchedPosts] = useState(null);
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState();

  const fetchPosts = useCallback(async function fetchPosts() {
    setIsLoading(fetchedPosts ? false : true);

    try {
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      setIsLoading(false);
      setError(null);
      setFetchedPosts(posts);
    } catch (error) {
      setIsLoading(false);
      setError(error.message);
      setFetchedPosts(null);
    }
  }, []);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );

  return (
    <>
      {isLoading && <p>Loading...</p>}
      {error && <p>{error}</p>}
      {fetchedPosts && <BlogPosts posts={fetchedPosts} />}
    </>
  );
}
```

Note

You'll find the complete example code on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/11-complex-state/examples/04-complex-usestate`.

When initiating the request, an `isLoading` state value (responsible for showing a loading indicator on the screen) should be set to `true` only if no data was fetched before. If data was fetched before (i.e., `fetchedPosts` is not `null`), that data should still be shown on the screen, instead of some loading indicator.

At first sight, this code might not look problematic. But it actually violates an important rule related to `useState()`: you should not reference the current state to set a new state value. If you need to do so, you should instead use the function form of the state updating function (see the *Updating State Based on Previous State Correctly* section of *Chapter 4*, *Working with Events and State*).

However, in the preceding example, this solution won't work. If you switch to the functional state-updating form, you only get access to the current value of the state you're trying to update. You don't get (safe) access to the current value of some other state. In the preceding example, another state (`fetchedPosts` instead of `isLoading`) is referenced. Therefore, you must violate the mentioned rule.

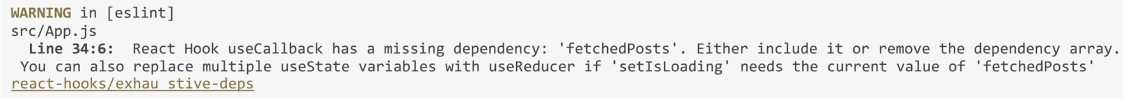
This violation also has real consequences (in this example). The following code snippet is part of a function called `fetchPosts`, which is wrapped with `useCallback()`:

```
const fetchPosts = useCallback(async function fetchPosts() {
  setIsLoading(fetchedPosts ? false : true);
  setError(null);

  try {
    const response = await fetch(
      'https://jsonplaceholder.typicode.com/posts'
    );

    if (!response.ok) {
      throw new Error('Failed to fetch posts.');
    }

    const posts = await response.json();

    setIsLoading(false);
    setError(null);
    setFetchedPosts(posts);
  } catch (error) {
    setIsLoading(false);
    setError(error.message);
    setFetchedPosts(null);
  }
}, []);
```

This function sends an HTTP request and changes multiple state values based on the state of the request.

`useCallback()` is used to avoid an infinite loop related to `useEffect()` (see *Chapter 8*, *Handling Side Effects*, to learn more about `useEffect()`, infinite loops, and `useCallback()` as a remedy). Normally, `fetchedPosts` should be added as a dependency to the `dependencies` array passed as a second argument to the `useCallback()` function. However, in this example, this can't be done because `fetchedPosts` is changed inside the function wrapped by `useCallback()`, and the state value is therefore not just a dependency but also actively changed. This causes an infinite loop.

As a result, a warning is shown in the terminal and the intended behavior of not showing the loading indicator if data was fetched before is not achieved:



```
WARNING in [eslint]
src/App.js
  Line 34:6:  React Hook useCallback has a missing dependency: 'fetchedPosts'. Either include it or remove the dependency array.
 You can also replace multiple useState variables with useReducer if 'setIsLoading' needs the current value of 'fetchedPosts'
react-hooks/exhau stive-deps
```

Figure 11.5: A warning about the missing dependency is output in the terminal

Problems like the one just described are common if you have multiple related state values that depend on each other.

One possible solution would be to move from multiple, individual state slices (`fetchedPosts`, `isLoading`, and `error`) to a single, combined state value (i.e., to an object). That would ensure that all state values are grouped together and can thereby be accessed safely when using the functional state-updating form. The state-updating code could then look like this:

```
setHttpState(prevState => ({
  fetchedPosts: prevState.fetchedPosts,
  isLoading: prevState.fetchedPosts ? false : true,
  error: null
}));
```

This solution would work. However, ending up with ever more complex (and nested) state objects, managed via `useState()`, is not typically desirable as it can make state management a bit harder and bloat your component code.

That's why React offers an alternative to `useState()`: the `useReducer()` Hook.

Managing State with useReducer( )

Just like `useState()`, `useReducer()` is a React Hook. And just like `useState()`, it is a Hook that can trigger component function re-evaluations. But, of course, it works slightly differently; otherwise, it would be a redundant Hook.

`useReducer()` is a Hook meant to be used for managing complex state objects. You will rarely (probably never) use it to manage simple string or number values.

This Hook takes two main arguments:

- A reducer function
- An initial state value

This brings up an important question: what is a reducer function?

Understanding Reducer Functions

In the context of `useReducer()`, a reducer function is a function that receives two parameters:

- The current state value
- An action that was dispatched

Besides receiving arguments, a reducer function must also return a value: the new state. It's called a reducer function because it reduces the old state (combined with an action) to a new state.

To make this all a bit easier to grasp and reason through, the following code snippet shows how `useReducer()` is used in conjunction with such a reducer function:

```
const initialHttpState = {
  data: null,
  isLoading: false,
  error: null,
};

function httpReducer(state, action) {
  if (action.type === 'FETCH_START') {
    return {
      ...state, // copying the existing state
      isLoading: state.data ? false : true,
      error: null,
    };
  }

  if (action.type === 'FETCH_ERROR') {
    return {
      data: null,
      isLoading: false,
      error: action.payload,
    };
  }

  if (action.type === 'FETCH_SUCCESS') {
    return {
      data: action.payload,
      isLoading: false,
      error: null,
    };
  }

  return initialHttpState; // default value for unknown actions
}

function App() {
  useReducer(httpReducer, initialHttpState);

  // more component code, not relevant for this snippet / explanation
}
```

At the bottom of this snippet, you can see that `useReducer()` is called inside of the `App` component function. Like all React Hooks, it must be called inside of component functions or other Hooks. You can also see the two arguments that were mentioned previously (the reducer function and the initial state value) being passed to `useReducer()`.

`httpReducer` is the reducer function. The function takes two arguments (`state`, which is the old state, and `action`, which is the dispatched action) and returns different state objects for different action types.

This reducer function takes care of all possible state updates. The entire state-updating logic is therefore outsourced from the component (note that `httpReducer` is defined outside of the component function).

But the component function must, of course, be able to trigger the defined state updates. That's where actions become important.

Note

In this example, the reducer function is created outside of the component function. You could also create it inside the component function, but that is not recommended. If you create the reducer function inside the component function, it will technically be recreated every time the component function is executed. This impacts performance unnecessarily since the reducer function does not need access to any component function values (state or props).

Dispatching Actions

The code shown previously is incomplete. When calling `useReducer()` in a component function, it does not just take two arguments. Instead, the Hook also returns a value—an array with exactly two elements (just like `useState()`, though the elements are different).

`useReducer()` should therefore be used like this (in the `App` component):

```
const [httpState, dispatch] = useReducer(
  httpReducer,
  initialHttpState
);
```

In this snippet, array destructuring is used to store the two elements (and it is always exactly two!) in two different constants: `httpState` and `dispatch`.

The first element in the returned array (`httpState`, in this case) is the state value returned by the reducer function. It's updated (meaning that the component function is called by React) whenever the reducer function is executed again. The element is called `httpState` in this example because it contains the state value, which is related to an HTTP request in this instance. That said, how you name the element in your case is up to you.

The second element (`dispatch`, in the example) is a function. It's a function that can be called to trigger a state update (i.e., to execute the reducer function again). When executed, the `dispatch` function must receive one argument—that is, the action value that will be available inside of the reducer function (via the reducer function's second argument). Here's how `dispatch` can be used in a component:

```
dispatch({ type: 'FETCH_START' });
```

The element is called `dispatch` in the example because it's a function used for dispatching actions to the reducer function. Just as before, the name is up to you, but `dispatch` is a commonly chosen name.

The shape and structure of that action value are also entirely up to you, but it's often set to an object that contains a `type` property. The `type` property is used in the reducer function to perform different actions for different types of actions. `type` therefore acts as an action identifier. You can see the `type` property being used inside the `httpReducer` function:

```
function httpReducer(state, action) {
  if (action.type === 'FETCH_START') {
    return {
      ...state, // copying the existing state
      isLoading: state.data ? false : true,
      error: null,
    };
  }

  if (action.type === 'FETCH_ERROR') {
    return {
      data: null,
      isLoading: false,
      error: action.payload,
    };
  }

  if (action.type === 'FETCH_SUCCESS') {
    return {
      data: action.payload,
      isLoading: false,
      error: null,
    };
  }

  return initialHttpState; // default value for unknown actions
}
```

You can add as many properties to the action object as needed. In the preceding example, some state updates access `action.payload` to extract some extra data from the action object. Inside a component, you would pass data along with the action like this:

```
dispatch({ type: 'FETCH_SUCCESS', payload: posts });
```

Again, the property name (`payload`) is up to you, but passing extra data along with the action allows you to perform state updates that rely on data generated by the component function.

Here's the complete, final code for the entire `App` component function:

```
// code for httpReducer etc. did not change

function App() {
  const [httpState, dispatch] = useReducer(
    httpReducer,
    initialHttpState
  );

  // Using useCallback() to prevent an infinite loop in useEffect()
  const fetchPosts = useCallback(async function fetchPosts() {
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, []);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );

  return (
    <>
      <header>
        <h1>Complex State Blog</h1>
        <button onClick={fetchPosts}>Load Posts</button>
      </header>
      {httpState.isLoading && <p>Loading...</p>}
      {httpState.error && <p>{httpState.error}</p>}
      {httpState.data && <BlogPosts posts={httpState.data} />}
    </>
  );
}
```

In this code snippet, you can see how different actions (with different `type` and sometimes `payload` properties) are dispatched. You can also see that the `httpState` value is used to show different user interface elements based on the state (e.g., `<p>Loading…</p>` is shown if `httpState.isLoading` is `true`).

Summary and Key Takeaways

- State management can have its challenges—especially when dealing with cross-component (or app-wide) state or complex state values.
- Cross-component state can be managed by lifting state up or by using React's context API.
- The context API is typically preferable if you do a lot of prop drilling (forwarding state values via props across multiple component layers).
- When using the context API, you use `createContext()` to create a new context object.
- The created context object is a component that must be wrapped around the part of the component tree that should get access to the context.
- When working with React 18 or older, the context object itself is not a component but an object that offers a nested `Provider` property that is a component.
- Components can access the context value via the `use()` (with React 19 or higher) or `useContext()` Hooks.
- For managing complex state values, `useReducer()` can be a good alternative to `useState()`.
- `useReducer()` utilizes a reducer function that converts the current state and a dispatched action to a new state value.
- `useReducer()` returns an array with exactly two elements: the state value and a dispatch function, which is used for dispatching actions.

What's Next?

Being able to manage both simple and complex state values efficiently is important. This chapter introduced two crucial tools that help with the task.

With the context API's `use()`, `useContext()`, and `useReducer()` Hooks, three new React Hooks were introduced. Combined with all the other Hooks covered thus far in the book, these mark the last of the React Hooks you will need in your everyday work as a React developer.

As a React developer, you're not limited to the built-in Hooks, though. You can also build your own Hooks. The next chapter will finally explore how that works and why you might want to build custom Hooks in the first place.

Test Your Knowledge!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to the examples that can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/11-complex-state/exercises/questions-answers.md`:

1. Which problem can be solved with React's context API?
2. Which three main steps have to be taken when using the context API?
3. When might `useReducer()` be preferred over `useState()`?
4. When working with `useReducer()`, what's the role of actions?

Apply What You Learned

Apply your knowledge about the context API and the `useReducer()` Hook to some real problems.

Activity 11.1: Migrating an App to the Context API

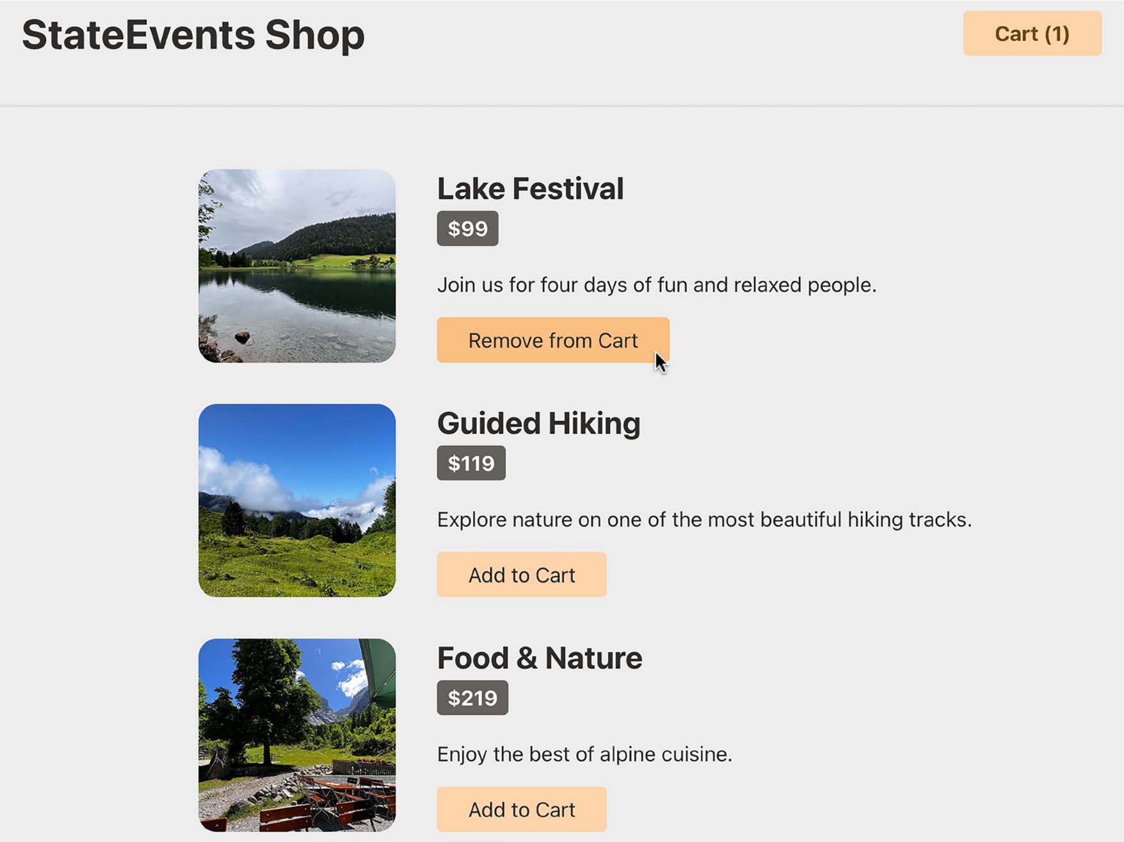
In this activity, your task is to improve an existing React project. Currently, the app is built without the context API, and so cross-component state is managed by lifting the state up. In this project, prop drilling is the consequence in some components. Therefore, the goal is to adjust the app such that the context API is used for cross-component state management.

Note

You can find the starting code for this activity at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/11-complex-state/activities/practice-1-start`. When downloading this code, you'll always download the entire repository. Make sure to then navigate to the subfolder with the starting code (`activities/practice-1-start` in this case) to use the right code snapshot.

The provided project also uses many features covered in earlier chapters. Take your time to analyze it and understand the provided code. This is great practice and allows you to see many key concepts in action.

Once you have downloaded the code and run `npm install` in the project folder (to install all required dependencies), you can start the development server via `npm run dev`. As a result, upon visiting `localhost:5173`, you should see the following user interface:



StateEvents Shop

Cart (1)

Lake Festival

$99

Join us for four days of fun and relaxed people.

Remove from Cart

Guided Hiking

$119

Explore nature on one of the most beautiful hiking tracks.

Add to Cart

Food & Nature

$219

Enjoy the best of alpine cuisine.

Add to Cart

Figure 11.6: The running starting project

To complete the activity, the steps are as follows:

1. Create a new context for the cart items.
2. Create a `Provider` component for the context and handle all context-related state changes there.
3. Provide the context (with the help of the `Provider` component) and make sure all components that need access to the context have access.
4. Remove the old logic (where state was lifted up).
5. Use the context in all the components that need access to it.

The user interface should be the same as that shown in *Figure 11.6* once you have completed the activity. Make sure that the user interface works exactly as it did before you implemented React's context features.

Note

All code files used for this activity, and the solution, can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/11-complex-state/activities/practice-1`.

Activity 11.2: Replacing useState() with useReducer()

In this activity, your task is to replace the `useState()` Hooks in the `Form` component with `useReducer()`. Use only one single reducer function (and thus only one `useReducer()` call) and merge all relevant state values into one state object.

Note

You can find the starting code for this activity at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/11-complex-state/activities/practice-2-start`. When downloading this code, you'll always download the entire repository. Make sure to then navigate to the subfolder with the starting code (`activities/practice-2-start` in this case) to use the right code snapshot.

The provided project also uses many features covered in earlier chapters. Take your time to analyze it and understand the provided code. This is great practice and allows you to see many key concepts in action.

Once you have downloaded the code and run `npm install` in the project folder (to install all required dependencies), you can start the development server via `npm run dev`. As a result, upon visiting `localhost:5173`, you should see the following user interface:

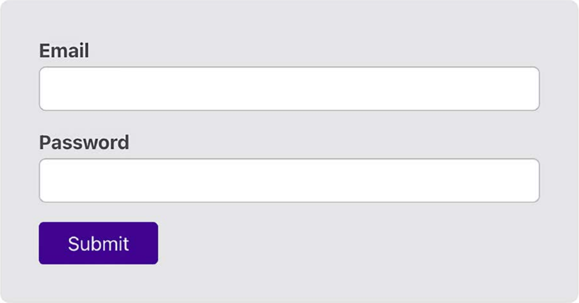


Figure 11.7: The running starting project

In the provided starting project, users get one of three results upon clicking the `Submit` button:

1. If one or both input fields didn't receive any input, an error message tells users to fill in the form.
2. If users entered values into both input fields, but at least one of the inputs holds an invalid value, a different error message is shown.

3. If users entered valid values into both input fields, the entered values are printed in the developer tools JavaScript console.

To complete the activity, the solution steps are as follows:

1. Remove (or comment out) the existing logic in the `Form` component that uses the `useState()` Hook for state management.
2. Add a reducer function that handles two actions (email changed and password changed) and also returns a default value.
3. Update the state object based on the dispatched action type (and payload, if needed).
4. Use the reducer function with the `useReducer()` Hook.
5. Dispatch the appropriate actions (with the appropriate data) in the `Form` component.
6. Use the state value where needed.

The user interface should be the same as that shown in *Figure 11.7* once you've finished the activity. Make sure that the user interface works exactly as it did before you implemented React's context features.

Note

All code files used for this activity, and the solution, can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/11-complex-state/activities/practice-2`.

12

Building Custom React Hooks

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Build your own React Hooks
- Use custom and default React Hooks in your components

Introduction

Throughout this book, one key React feature has been referenced repeatedly in many different variations. That feature is React Hooks.

Hooks power almost all core functionalities and concepts offered by React—from state management in a single component to accessing cross-component state (context) in multiple components. They enable you to access JSX elements via refs and allow you to handle side effects inside of component functions.

Without Hooks, modern React would not work, and building feature-rich applications would be impossible.

Thus far, only built-in Hooks have been introduced and used. However, you can build your own custom Hooks as well—or you can use custom Hooks built by other developers (e.g., by using third-party libraries). In this chapter, you will learn why you might want to do this and how it works.

Introducing Custom Hooks

Before starting to build custom Hooks, it's very important to understand what exactly custom Hooks are.

In React apps, custom Hooks are regular JavaScript functions that satisfy the following conditions:

- The function name starts with `use` (just as all built-in Hooks start with `use`: `useState()`, `useReducer()`, etc.).

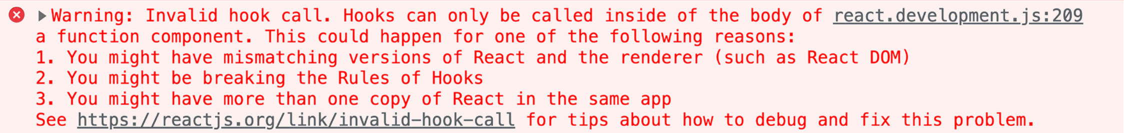
- The function calls another React Hook (a built-in one or a custom one—doesn't matter).
- The function does not just return JSX code (otherwise, it would essentially be a React component), though it could return some JSX code—as long as that's not the only value returned.

If a function meets these three conditions, it can (and should) be called a custom (React) Hook. So, custom Hooks are really just normal functions with special names (starting with `use`) that call other (custom or built-in) Hooks and that do not (just) return JSX code. If you try to call a (custom or built-in) Hook in some other place (e.g., outside of any function or in a regular, non-Hook function), you might get a warning (depending on your project setup; see below).

For example, the following function uses the `useEffect()` Hook but has a name that does not start with `use`. It is therefore not in line with the official naming recommendation:

```
function sendAnalyticsEvent(event) {
  useEffect(() => {
    fetch('https://my-analytics-backend.com/events', {
      method: 'POST',
      body: JSON.stringify(event)
    })
  }, []);
}
```

In projects that perform code linting to check your code for rule violations, this code would produce a warning because this function doesn't qualify as a custom Hook (due to its name).



```
▶Warning: Invalid hook call. Hooks can only be called inside of the body of react.development.js:209
a function component. This could happen for one of the following reasons:
1. You might have mismatching versions of React and the renderer (such as React DOM)
2. You might be breaking the Rules of Hooks
3. You might have more than one copy of React in the same app
See https://reactjs.org/link/invalid-hook-call for tips about how to debug and fix this problem.
```

Figure 12.1: React complains if you call a Hook function in the wrong place

As the warning states, Hooks, whether custom or built-in, must only be called inside component functions. And, even though the warning message doesn't explicitly mention it, they may also be called inside of custom Hooks.

So, if the `sendAnalyticsEvent()` function is renamed `useSendAnalyticsEvent()`, the warning disappears since the function now qualifies as a custom Hook.

Even though it's technically not a hard rule that's enforced by React itself, it's a strong recommendation to follow this naming convention.

Being able to build custom Hooks is an extremely important feature because it means that you can build reusable non-component functions that can contain state logic (via `useState()` or `useReducer()`), handle side effects in your reusable custom Hook functions (via `useEffect()`), or use any other React Hook. With normal, non-Hook functions, none of these would be possible, and you would therefore be unable to outsource any logic that involves a React Hook into such functions.

In this way, custom Hooks complement the concept of React components. While React components are reusable UI building blocks (which may contain stateful logic), custom Hooks are reusable logic snippets that can be used in your component functions. Thus, custom Hooks help you reuse shared logic across components. For example, custom Hooks enable you to outsource the logic for sending an HTTP request and handling the related states (loading, error, etc.).

Why Would You Build Custom Hooks?

In the previous chapter (*Chapter 11*, *Working with Complex State*), when the `useReducer()` Hook was introduced, an example was provided in which the Hook was utilized in sending an HTTP request. Here's the relevant, final code again:

```
const initialHttpState = {
  data: null,
  isLoading: false,
  error: null,
};

function httpReducer(state, action) {
  if (action.type === 'FETCH_START') {
    return {
      ...state, // copying the existing state
      isLoading: state.data ? false : true,
      error: null,
    };
  }

  if (action.type === 'FETCH_ERROR') {
    return {
      data: null,
      isLoading: false,
      error: action.payload,
    };
  }

  if (action.type === 'FETCH_SUCCESS') {
    return {
      data: action.payload,
      isLoading: false,
      error: null,
    };
  }
  return initialHttpState; // default value for unknown actions
}

function App() {
  const [httpState, dispatch] = useReducer(
    httpReducer,
    initialHttpState
  );

  const fetchPosts = useCallback(async function fetchPosts() {
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, []);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );

  return (
    <>
      <header>
        <h1>Complex State Blog</h1>
        <button onClick={fetchPosts}>Load Posts</button>
      </header>
      {httpState.isLoading && <p>Loading...</p>}
      {httpState.error && <p>{httpState.error}</p>}
      {httpState.data && <BlogPosts posts={httpState.data} />}
    </>
  );
};
```

In this code example, an HTTP request is sent whenever the `App` component is rendered for the first time. The HTTP request fetches a list of (dummy) posts. Until the request finishes, a loading message (`<p>Loading…</p>`) is displayed to the user. If there's an error, an error message is displayed.

As you can see, quite a lot of code must be written to handle this relatively basic use case. And, especially in bigger React apps, it is quite likely that multiple components will need to send HTTP requests. They probably won't need to send the exact same request to the same URL (`https://jsonplaceholder.typicode.com/posts`, in this example), but it's definitely possible that different components will fetch different data from different URLs.

Therefore, almost the exact same code must be written over and over again in multiple components. And it's not just the code for sending the HTTP request (i.e., the function wrapped by `useCallback()`). Instead, the HTTP-related state management (done via `useReducer()`, in this example), as well as the request initialization via `useEffect()`, must be repeated in all those components.

And that is where custom Hooks come in to save the day. Custom Hooks help you avoid this repetition by allowing you to build reusable, potentially stateful "logic snippets" that can be shared across components.

A First Custom Hook

Before exploring advanced scenarios and solving the HTTP request problem mentioned previously, here's a more basic example of a first, custom Hook:

```
import { useState } from 'react';

function useCounter() {
  const [counter, setCounter] = useState(0);

  function increment() {
    setCounter(oldCounter => oldCounter + 1);
  };

  function decrement() {
    setCounter(oldCounter => oldCounter - 1);
  };
  return { counter, increment, decrement };
};

export default useCounter;
```

This code can be stored in a file named `use-counter.js` inside a `hooks/` folder—though both names are totally up to you. There are no rules regarding the file or the folder name (or, in general, the place where you store this code). The file extension is `.js` instead of `.jsx` since this file contains no JSX code.

As you can see, `useCounter` is a regular JavaScript function. The name of the function starts with `use`, and therefore this function qualifies as a custom Hook (meaning you won't get any warning messages when using other Hooks inside of it).

Inside `useCounter()`, a `counter` state is managed via `useState()`. The state is changed via two nested functions (`increment` and `decrement`), and the state, as well as the functions, is returned by `useCounter` (grouped together in a JavaScript object).

Note

The syntax used to group `counter`, `increment`, and `decrement` together uses a regular JavaScript feature: shorthand property names.

If a property name in an object literally matches the name of the variable whose value is assigned to the property, you can use this shorter notation.

Instead of writing `{ counter: counter, increment: increment, decrement: decrement }`, you can use the shorthand notation `{ counter, increment, decrement }` shown in the snippet above.

This custom Hook can be stored in a separate file (e.g., in a `hooks` folder inside the React project, such as `src/hooks/use-counter.js`). Thereafter, it can be used in any React component, and you can use it in as many React components as needed.

For example, the following two components (`Demo1` and `Demo2`) could use this `useCounter` Hook as follows:

```
import useCounter from './hooks/use-counter.js';

function Demo1() {
  const { counter, increment, decrement } = useCounter();

  return (
    <>
      <p>{counter}</p>
      <button onClick={increment}>Inc</button>
      <button onClick={decrement}>Dec</button>

    </>
  );
};

function Demo2() {
  const { counter, increment, decrement } = useCounter();

  return (
    <>
      <p>{counter}</p>
      <button onClick={increment}>Inc</button>
      <button onClick={decrement}>Dec</button>
    </>
  );
};

function App() {
  return (
    <main>
      <Demo1 />
      <Demo2 />
    </main>
  );
};

export default App;
```

Note

You will find the full example code at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/12-custom-hooks/examples/01-first-hook`.

The `Demo1` and `Demo2` components both execute `useCounter()` inside of their component functions. The `useCounter()` function is called a normal function because it is a regular JavaScript function.

Since the `useCounter` Hook returns an object with three properties (`counter`, `increment`, and `decrement`), `Demo1` and `Demo2` use object destructuring to store the property values in local constants. These values are then used in the JSX code to output the `counter` value and connect the two `<button>` elements to the `increment` and `decrement` functions.

After pressing the buttons a couple of times each, the resulting user interface might look like this:

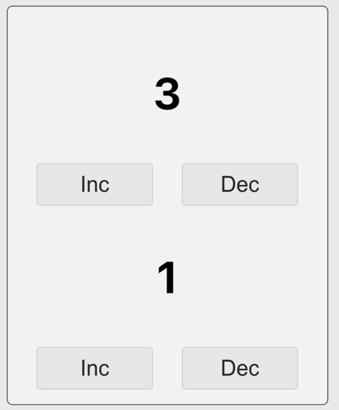


Figure 12.2: Two independent counters

In this screenshot, you can also see a very interesting and important behavior of custom Hooks. That is, if the same stateful custom Hook is used in multiple components, every component gets its own state. The `counter` state is not shared. The `Demo1` component manages its own `counter` state (through the `useCounter()` custom Hook), and so does the `Demo2` component.

Custom Hooks: A Flexible Feature

The two independent states of `Demo1` and `Demo2` show a very important feature of custom Hooks: you use them to share logic, not state. If you needed to share state across components, you would do so with React context (see the previous chapter).

When using Hooks, every component uses its own "instance" (or "version") of that Hook. It's always the same logic, but any state or side effects handled by a Hook are handled on a per-component basis.

It's also worth noting that custom Hooks **can** be stateful but **don't have to be**. They can manage state via `useState()` or `useReducer()`, but you could also build custom Hooks that only handle side effects (without any state management).

There's only one thing you implicitly have to do in custom Hooks: you must use some other React Hook (custom or built-in). This is because if you didn't include any other Hook, there would be no need to build a custom Hook in the first place. A custom Hook is just a regular JavaScript function (with a name starting with `use`) with which you are allowed to use other Hooks. If you don't need to use any other Hooks, you can simply build a normal JavaScript function with a name that does not start with `use`.

You also have a lot of flexibility regarding the logic inside the Hook, its parameters, and the value it returns. Regarding the Hook logic, you can add as much logic as needed. You can manage no state or multiple state values. You can include other custom Hooks or only use built-in Hooks. You can manage multiple side effects, work with refs, or perform complex calculations. There are no restrictions regarding what can be done in a custom Hook.

Custom Hooks and Parameters

You can also accept and use parameters in your custom Hook functions. For example, the `useCounter` Hook from the *A First Custom Hook* section can be adjusted to take an initial counter value and separate values by which the counter should be increased or decreased, as shown in the following snippet:

```
import { useState } from 'react';

function useCounter(initialValue, incVal, decVal) {
  const [counter, setCounter] = useState(initialValue);

  function increment() {
    setCounter(oldCounter => oldCounter + incVal);
  };

  function decrement() {
    setCounter(oldCounter => oldCounter - decVal);
  };

  return { counter, increment, decrement };
};

export default useCounter;
```

In this adjusted example, the `initialValue` parameter is used to set the initial state via `useState(initialValue)`. The `incVal` and `decVal` parameters are used in the `increment` and `decrement` functions to change the `counter` state with different values.

Of course, once parameters are used in a custom Hook, fitting parameter values must be provided when the custom Hook is called in a component function (or in another custom Hook). Therefore, the code for the `Demo1` and `Demo2` components must also be adjusted—for example, like this:

```
function Demo1() {
  const { counter, increment, decrement } = useCounter(1, 2, 1);

  return (
    <>
      <p>{counter}</p>
      <button onClick={increment}>Inc</button>
      <button onClick={decrement}>Dec</button>

    </>
  );
};
```

```
function Demo2() {
  const { counter, increment, decrement } = useCounter(0, 1, 2);

  return (
    <>
      <p>{counter}</p>
      <button onClick={increment}>Inc</button>
      <button onClick={decrement}>Dec</button>
    </>
  );
};
```

Note

You can also find this code on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/12-custom-hooks/examples/02-parameters`.

Now, both components pass different parameter values to the `useCounter` Hook function. Therefore, they can reuse the same Hook and its internal logic dynamically.

Custom Hooks and Return Values

As shown with `useCounter`, custom Hooks may return values. And this is important: they **may** return values, but they don't have to. If you build a custom Hook that only handles some side effects (via `useEffect()`), you don't have to return any value (because there probably isn't any value that should be returned).

But if you do need to return a value, you decide which type of value you want to return. You could return a single number or string. If your Hook must return multiple values (like `useCounter` does), you can group these values into an array or object. You can also return arrays that contain objects or vice versa. In short, you can return anything. It is a normal JavaScript function, after all.

Some built-in Hooks such as `useState()` and `useReducer()` return arrays (with a fixed number of elements). `useRef()`, on the other hand, returns an object (which always has a `current` property). `useEffect()` returns nothing. Your Hooks can therefore return whatever you want.

For example, the `useCounter` Hook from previously could be rewritten to return an array instead:

```
import { useState } from 'react';

function useCounter(initialValue, incVal, decVal) {
  const [counter, setCounter] = useState(initialValue);

  function increment() {
    setCounter((oldCounter) => oldCounter + incVal);
  }

  function decrement() {
    setCounter((oldCounter) => oldCounter - decVal);
  }

  return [counter, increment, decrement];
}

export default useCounter;
```

To use the returned values, then, the `Demo1` and `Demo2` components need to switch from object destructuring to array destructuring, as follows:

```
function Demo1() {
  const [counter, increment, decrement] = useCounter(1, 2, 1);

  return (
    <>
      <p>{counter}</p>
      <button onClick={increment}>Inc</button>
      <button onClick={decrement}>Dec</button>
    </>
  );
}

function Demo2() {
  const [counter, increment, decrement] = useCounter(0, 1, 2);

  return (
    <>
      <p>{counter}</p>
      <button onClick={increment}>Inc</button>
      <button onClick={decrement}>Dec</button>
    </>
  );
}
```

The two components behave like before, so you can decide which return value you prefer.

Note

This finished code can also be found on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/12-custom-hooks/examples/03-return-values`.

A More Complex Example

The previous examples were deliberately rather simple. Now that the basics of custom Hooks are clear, it makes sense to dive into a slightly more advanced and realistic example.

Consider the HTTP request example from the beginning of this chapter:

```
const initialHttpState = {
  data: null,
  isLoading: false,
  error: null,
};

function httpReducer(state, action) {
  if (action.type === 'FETCH_START') {
    return {
      ...state, // copying the existing state
      isLoading: state.data ? false : true,
      error: null,
    };
  }

  if (action.type === 'FETCH_ERROR') {
    return {
      data: null,
      isLoading: false,
      error: action.payload,
    };
  }

  if (action.type === 'FETCH_SUCCESS') {
    return {
      data: action.payload,
      isLoading: false,
      error: null,
    };
  }
  return initialHttpState; // default value for unknown actions
}

function App() {
  const [httpState, dispatch] = useReducer(
    httpReducer,
    initialHttpState
  );

  const fetchPosts = useCallback(async function fetchPosts() {
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, []);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );

  return (
    <>
      <header>
        <h1>Complex State Blog</h1>
        <button onClick={fetchPosts}>Load Posts</button>
      </header>
      {httpState.isLoading && <p>Loading...</p>}
      {httpState.error && <p>{httpState.error}</p>}
      {httpState.data && <BlogPosts posts={httpState.data} />}
    </>
  );
};
```

In that example, the entire `useReducer()` logic (including the reducer function, `httpReducer`) and the `useEffect()` call can be outsourced into a custom Hook. The result would be a very lean `App` component and a reusable Hook that could be used in other components as well.

Building a First Version of the Custom Hook

This custom Hook could be named `useFetch` (since it fetches data), and it could be stored in `hooks/use-fetch.js`. Of course, both the Hook name as well as the file storage path are up to you. Here's how the first version of `useFetch` might look:

```
import { useCallback, useEffect, useReducer } from 'react';

const initialHttpState = {
  data: null,
  isLoading: false,
  error: null,
};

function httpReducer(state, action) {
  // same reducer code as before
}

function useFetch() {
  const [httpState, dispatch] = useReducer(
    httpReducer,
    initialHttpState
  );

  const fetchPosts = useCallback(async function fetchPosts() {
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, []);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );
}

export default useFetch;
```

Please note that this is not the final version.

In this first version, the `useFetch` Hook contains the `useReducer()` and `useEffect()` logic. It's worth noting that the `httpReducer` function is created outside of `useFetch`. This ensures that the function is not recreated unnecessarily when `useFetch()` is re-executed (which will happen often as it is called every time the component that uses this Hook is re-evaluated). The `httpReducer` function will therefore only be created once (for the entire application lifetime), and that same function instance will be shared by all components that use `useFetch`.

Since `httpReducer` is a pure function (that is, it always produces new return values that are based purely on the parameter values), sharing this function instance is fine and won't cause any unexpected bugs. If `httpReducer` were to store or manipulate any values that are not based on function inputs, it should be created inside of `useFetch` instead. This way, you avoid having multiple components accidentally manipulate and use shared values.

However, this version of the `useFetch` Hook has two big issues:

- Currently, no value is returned. Therefore, components that use this Hook won't get access to the fetched data or the loading state.

- The HTTP request URL is hardcoded into `useFetch`. As a result, all components that use this Hook will send the same kind of request to the same URL.

Therefore, to improve this Hook, these two issues must be tackled—starting with the first one.

Making the Hook Useful by Returning Values

The first issue can be solved by returning the fetched data (or `undefined`, if no data was fetched yet), the loading state value, and the error value. Since these values are exactly the values that make up the `httpState` object returned by `useReducer()`, `useFetch` can simply return that entire `httpState` object, as shown here:

```
// httpReducer function and initial state did not change,
// hence omitted here
function useFetch() {
  const [httpState, dispatch] = useReducer(
    httpReducer,
    initialHttpState
  );

  const fetchPosts = useCallback(async function fetchPosts() {
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, []);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );

  return httpState;
}
```

The only thing that changed in this code snippet is the last line of the `useFetch` function. With `return httpState`, the state managed by `useReducer()` (and therefore by the `httpReducer` function) is returned by the custom Hook.

With that first issue fixed, the next step is to also make the Hook more reusable.

Improving Reusability by Accepting an Input Parameter

To fix the second problem (i.e., the hardcoded URL), a parameter should be added to `useFetch`:

```
// httpReducer function and initial state did not change, hence omitted here
function useFetch(url) {
  const [httpState, dispatch] = useReducer(
    httpReducer,
    initialHttpState
  );

  const fetchPosts = useCallback(async function fetchPosts() {
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(url);

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, [url]);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );

  return httpState;
}
```

In this snippet, the `url` parameter was added to `useFetch`. This parameter value is then used inside the `try` block when calling `fetch(url)`. Please note that `url` was also added as a dependency to the `useCallback()` dependencies array.

Since `useCallback()` is wrapped around the fetching function (to prevent infinite loops by `useEffect()`), any external values used inside of `useCallback()` must be added to its dependencies array. Since `url` is an external value (meaning it's not defined inside of the wrapped function), it must be added. This also makes sense logically: if the `url` parameter were to change (i.e., if the component that uses `useFetch` changes it), a new HTTP request should be sent.

This final version of the `useFetch` Hook can now be used in all components to send HTTP requests to different URLs and use the HTTP state values as needed by the components.

For example, the `App` component can use `useFetch` like this:

```
import BlogPosts from './components/BlogPosts.jsx';
import useFetch from './hooks/use-fetch.js';

function App() {
  const { data, isLoading, error } = useFetch(
    'https://jsonplaceholder.typicode.com/posts'
  );

  return (
    <>
      <header>
        <h1>Complex State Blog</h1>
      </header>
      {isLoading && <p>Loading...</p>}
      {error && <p>{error}</p>}
      {data && <BlogPosts posts={data} />}
    </>
  );
}

export default App;
```

The component imports and calls `useFetch()` (with the appropriate URL as an argument) and uses object destructuring to get the `data`, `isLoading`, and `error` properties from the `httpState` object. These values are then used in the JSX code.

Of course, the `useFetch` Hook could also return a pointer to the `fetchPosts` function (in addition to `httpState`) to allow components such as the `App` component to manually trigger a new request, as shown here:

```
// httpReducer function and initial state did not change, hence omitted here
function useFetch(url) {
  const [httpState, dispatch] = useReducer(
    httpReducer,
    initialHttpState
   );

  const fetchPosts = useCallback(async function fetchPosts() {
    dispatch({ type: 'FETCH_START' });

    try {
      const response = await fetch(url);

      if (!response.ok) {
        throw new Error('Failed to fetch posts.');
      }

      const posts = await response.json();

      dispatch({ type: 'FETCH_SUCCESS', payload: posts });
    } catch (error) {
      dispatch({ type: 'FETCH_ERROR', payload: error.message });
    }
  }, [url]);

  useEffect(
    function () {
      fetchPosts();
    },
    [fetchPosts]
  );

  return [ httpState, fetchPosts ];
}
```

In this example, the `return` statement was changed. Instead of returning just `httpState`, `useFetch` now returns an array that contains the `httpState` object and a pointer to the `fetchPosts` function. Alternatively, `httpState` and `fetchPosts` could have been merged into an object (instead of an array).

In the `App` component, `useFetch` could now be used like this:

```
import BlogPosts from './components/BlogPosts.jsx';
import useFetch from './hooks/use-fetch.js';

function App() {
  const [{ data, isLoading, error }, fetchPosts] = useFetch(
    'https://jsonplaceholder.typicode.com/posts'
  );

  return (
    <>
      <header>
        <h1>Complex State Blog</h1>
        <button onClick={fetchPosts}>Load Posts</button>
      </header>
      {isLoading && <p>Loading...</p>}
      {error && <p>{error}</p>}
      {data && <BlogPosts posts={data} />}
    </>
  );
}

export default App;
```

The `App` component uses array and object destructuring combined to extract the returned values (and the values nested in the `httpState` object). A newly added `<button>` element is then used to trigger the `fetchPosts` function.

This example effectively shows how custom Hooks can lead to much leaner component functions by allowing easy logic reuse, with or without state or side effects.

In addition, Hooks can also enable some interesting patterns—for example, related to React's Context API.

Using Custom Hooks for Context Access

As hinted in the previous chapter, in the *Outsourcing Context Logic into Separate Components* section, you can use custom Hooks to improve the process of consuming context values in components.

For example, if you provide some context named `BookmarkContext` (e.g., via a `<BookmarkContextProvider>` component), you can access this context value inside components like this:

```
import { use } from 'react';

import BookmarkContext from '../../store/bookmark-context.jsx';

function BookmarkSummary() {
  const bookmarkCtx = use(BookmarkContext);

  // other component code, including returned JSX code
}
```

However, instead of directly accessing the context value like this, you could also build the following custom Hook (e.g., stored in a `store/use-bookmark-context.js` file):

```
import { use } from 'react';

import BookmarkContext from './bookmark-context.jsx';

function useBookmarkContext() {
  const bookmarkCtx = use(BookmarkContext);

  return bookmarkCtx;
}
export default useBookmarkContext;
```

But, of course, this Hook doesn't really provide any advantages compared to directly consuming the context value in a component via `use()`.

That changes once you enrich this custom Hook with more useful logic—for example, with error handling if it's used in a place where the context is not available:

```
function useBookmarkContext() {
  const bookmarkCtx = use(BookmarkContext);

  if(!bookmarkCtx) {
    throw new Error('BookmarkContext must be provided!')
  }

  return bookmarkCtx;
}
```

This Hook can then be used in your components to get hold of the context value like this:

```
import useBookmarkContext from '../../store/use-bookmark-context.js';

function BookmarkSummary() {
  const bookmarkCtx = useBookmarkContext();

  // other component code, including returned JSX code
}
```

This therefore is not just another example of a custom Hook, but also a common pattern you should know. It's a pattern that's used in many React projects since it ensures that you don't accidentally try to use the context value in a place where it's not accessible (i.e., in a component that's not wrapped by `BookmarkContextProvider`).

Of course, it's not a pattern you must use, though. But it's something you could consider using to get an early error if you're trying to access your context in the wrong place. If you're distributing a library that exposes some context, it's an especially helpful pattern since it warns your library users in case they forget to provide the context.

Summary and Key Takeaways

- You can create custom Hooks to outsource and reuse logic that relies on other built-in or custom Hooks.
- Custom Hooks are regular JavaScript functions with names that start with `use`.
- Custom Hooks can call any other Hooks.
- Therefore, custom Hooks can, for example, manage state or perform side effects.
- All components can use custom Hooks by simply calling them like any other (built-in) Hooks.
- When multiple components use the same custom Hook, every component receives its own "instance" (i.e., its own state value, etc.).
- Inside custom Hooks, you can accept any parameter values and return any values of your choice.

What's Next?

Custom Hooks are a key React feature since they help you to write leaner components and reuse (stateful) logic across them. Especially when building more complex React apps (consisting of dozens or even hundreds of components), custom Hooks can lead to tremendously more manageable code.

Combined with components, props, state (via `useState()` or `useReducer()`), side effects, and all the other concepts covered in this and previous chapters, you now have a very solid foundation that allows you to build production-ready React apps. Therefore, you're now prepared to dive into more advanced React concepts as well as crucial third-party packages that you should know about.

For example, most React apps don't just consist of one single page—instead, at least on most websites, users should be able to switch between multiple pages. For example, an online shop has a list of products, product detail pages, a shopping cart page, and many other pages.

The next chapter will therefore explore how you can build such multipage apps with React and the popular React Router third-party package.

Test Your Knowledge!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to examples that can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/12-custom-hooks/exercises/questions-answers.md`:

1. What is the definition of a custom Hook?
2. Which special feature can be used inside a custom Hook?
3. What happens when multiple components use the same custom Hook?
4. How can custom Hooks be made more reusable?

Apply What You Learned

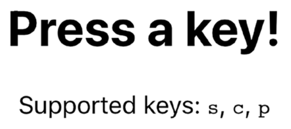
Apply your knowledge about custom Hooks.

Activity 12.1: Build a Custom Keyboard Input Hook

In this activity, your task is to refactor a provided component such that it's leaner and no longer contains any state or side-effect logic. Instead, you should create a custom Hook that contains that logic. This Hook could then potentially be used in other areas of the React application as well.

> **Note**
>
> You can find the starting code for this activity at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/12-custom-hooks/activities/practice-1-start`. When downloading this code, you'll always download the entire repository. Make sure to then navigate to the subfolder with the starting code (`activities/practice-1-start`, in this case) to use the right code snapshot.
>
> The provided project also uses many features covered in earlier chapters. Take your time to analyze it and understand the provided code. This is a great practice and allows you to see many key concepts in action.

Once you have downloaded the code and run `npm install` in the project folder to install all required dependencies, you can start the development server via `npm run dev`. As a result, upon visiting `localhost:5173`, you should see the following user interface:



Press a key!

Supported keys: `s`, `c`, `p`

Figure 12.3: The running starting project

To complete the activity, the solution steps are as follows:

1. Create a new custom Hook file (e.g., in the `src/hooks` folder) and create a Hook function in that file.
2. Move the side effect and state management logic into that new Hook function.
3. Make the custom Hook more reusable by accepting and using a parameter that controls which keys are allowed.
4. Return the state managed by the custom Hook.
5. Use the custom Hook and its returned value in the `App` component.

The user interface should be the same once you have completed the activity, but the code of the `App` component should change. After finishing the activity, `App` should contain only this code:

```
function App() {
  const pressedKey = useKeyEvent(['s', 'c', 'p']); // this is your Hook!

  let output = '';

  if (pressedKey === 's') {
    output = '';
  } else if (pressedKey === 'c') {
    output = '';
  } else if (pressedKey === 'p') {
    output = '';
  }

  return (
    <main>
      <h1>Press a key!</h1>
      <p>
        Supported keys: <kbd>s</kbd>, <kbd>c</kbd>, <kbd>p</kbd>
      </p>
      <p id="output">{output}</p>
    </main>
  );
}
```

Note

All code files used for this activity, and an example solution, can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/12-custom-hooks/activities/practice-1`.

13

Multipage Apps with React Router

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Build multipage single-page applications (and understand why this is not an oxymoron)
- Use the React Router package to load different React components for different URL paths
- Create static and dynamic routes (and understand what routes are in the first place)
- Navigate the website via both links and programmatic commands
- Build nested page layouts

Introduction

Having worked through the first twelve chapters of this book, you should now know how to build React components and web apps, as well as how to manage components and app-wide state, and how to share data between components (via props or context).

But even though you know how to compose a React website from multiple components, all these components are on the same single website page. Sure, you can display components and content conditionally, but users will never switch to a different page. This means that the URL path will never change; users will always stay on `your-domain.com`. Also, at this point in time, your React apps don't support any paths such as `your-domain.com/products` or `your-domain.com/blog/latest`.

Note

Uniform Resource Locators (**URLs**) are references to web resources. For example, `https://academind.com/courses` is a URL that points to a specific page of the author's website. In this example, `academind.com` is the **domain name** of the website and `/courses` is the **path** to a specific website page.

For React apps, it might make sense that the path of the loaded website never changes. After all, in *Chapter 1*, *React – What and Why*, you learned that you build **single-page applications** (**SPAs**) with React.

But even though it might make sense, it's also quite a serious limitation.

One Page Is Not Enough

Having just a single page means that complex websites that would typically consist of multiple pages (e.g., an online shop with pages for products, orders, and more) become quite difficult to build with React. Without multiple pages, you have to fall back to state and conditional values to display different content on the screen.

But without changing URL paths, your website visitors can't share links to anything but the starting page of your website. Also, any conditionally loaded content will be lost when a new visitor visits that starting page. That will also be the case if users simply reload the page they're currently on. A reload fetches a new version of the page, and so any state (and therefore user interface) changes are lost.

For these reasons, you absolutely need a way of including multiple pages (with different URL paths) in a single React app for most React websites. Thanks to modern browser features and a highly popular third-party package, that is indeed possible (and the default for most React apps).

Via the **React Router** package, your React app can listen to URL path changes and display different components for different paths. For example, you could define the following path-component mappings:

- `<domain>/` => `<Home />` component is loaded.
- `<domain>/products` => `<ProductList />` component is loaded.
- `<domain>/products/p1` => `<ProductDetail />` component is loaded.
- `<domain>/about` => `<AboutUs />` component is loaded.

Technically, it will still be a SPA because there's still only one HTML page being sent to website users. But in that single-page React app, different components are rendered conditionally by the React Router package based on the specific URL paths that are being visited. As the developer of the app, you don't have to manually manage this kind of state or render content conditionally—React Router will do it for you. In addition, your website is able to handle different URL paths, and therefore, individual pages can be shared or reloaded.

Getting Started with React Router and Defining Routes

React Router is a third-party React library that can be installed in any React project. Once installed, you can use various components in your code to enable the aforementioned features.

Inside your React project, the package is installed via this command:

```
npm install react-router-dom
```

Once installed, you can import and use various components (and Hooks) from that library.

To start supporting multiple pages in your React app, you need to set up **routing** by going through the following steps:

1. Create different components for your different pages (e.g., `Dashboard` and `Orders` components).
2. Use the `createBrowserRouter()` function and the `RouterProvider` component from the React Router library to enable routing and define the **routes** that should be supported by the React app.

In this context, the term **routing** refers to the React app being able to load different components for different URL paths (e.g., different components for the `/` and `/orders` paths). A route is a definition that's added to the React app that defines the URL path for which a predefined JSX snippet should be rendered (e.g., the `Orders` component should be loaded for the `/orders` path).

In an example React app that contains `Dashboard` and `Orders` components, and wherein the React Router library was installed via `npm install`, you can enable routing and navigation between these two components by editing the root component (in `src/App.jsx`) like this:

```
import {
  createBrowserRouter,
  RouterProvider
} from 'react-router-dom';

import Dashboard from './routes/Dashboard.jsx';
import Orders from './routes/Orders.jsx';

const router = createBrowserRouter([
  { path: '/', element: <Dashboard /> },
  { path: '/orders', element: <Orders /> }
]);

function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

Note

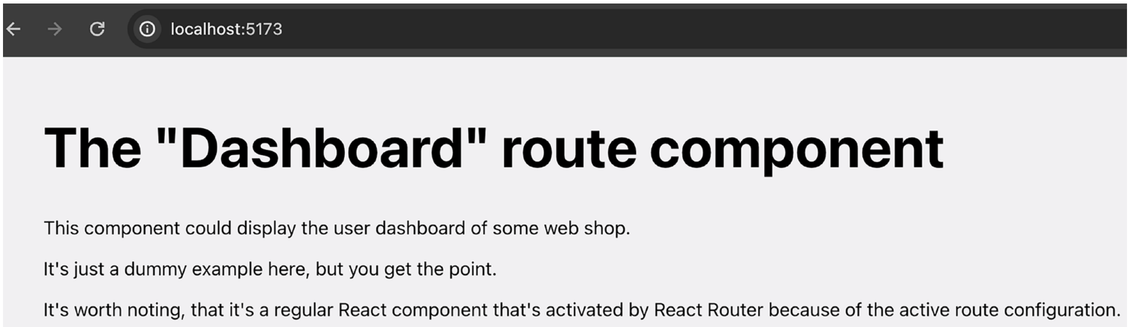
You can find the complete example code on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/examples/01-getting-started-with-routing`.

In the preceding code snippet, React Router's `createBrowserRouter()` function is called to create a `router` object that contains the application's route configuration (a list of available routes). The array passed to `createBrowserRouter()` contains route definition objects, where every object defines a `path` for which the route should be matched and an `element` that should be rendered.

React Router's `RouterProvider` component is then used to set the `router` configuration and define a place for the active route elements to be rendered.

You can think of the `<RouterProvider />` element being replaced with the content defined via the `element` property once a route becomes active. Therefore, the positioning of the `RouterProvider` component matters. In this case (and probably in most React apps), it's the root application component—i.e., React Router, that should control the entire application component tree.

If you run the provided example React app (via `npm run dev`), you'll see the following output on the screen:



The "Dashboard" route component

This component could display the user dashboard of some web shop.

It's just a dummy example here, but you get the point.

It's worth noting, that it's a regular React component that's activated by React Router because of the active route configuration.

Figure 13.1: The Dashboard component content is loaded

The content of the `Dashboard` component is displayed on the screen if you visit `localhost:5173`. Please note that the visible page content is not defined in the `App` component (in the code snippet shared previously). Instead, only two route definitions were added: one for the `/` path (i.e., for `localhost:5173/` or just `localhost:5173`, without the trailing forward slash—it's handled in the same way) and one for the `/orders` path (`localhost:5173/orders`).

Note

`localhost` is a local address that's typically used for development. When you deploy your React app (i.e., you upload it to a web server), you will receive a different domain—or assign a custom domain. Either way, it will not be `localhost` after deployment.

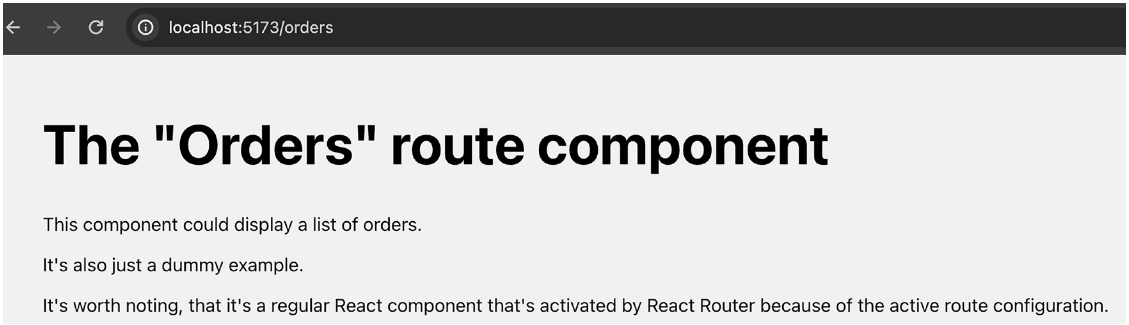
The part after `localhost` (`:5173`) defines the network port to which the request will be sent. Without the additional port information, ports `80` or `443` (as the default HTTP(S) ports) are used automatically. During development, however, these are not the ports you want. Instead, you would typically use ports such as `5173`, `8000`, or `8080` as these are normally unoccupied by any other system processes and hence can be used safely. Projects created via Vite typically use port `5173`.

Since `localhost:5173` is loaded by default (when running `npm run dev`), the first route definition (`{ path: '/', element: <Dashboard /> }`) becomes active. This route is active because its path (`'/'`) matches the path of `localhost:5173` (since this is the same as `localhost:5173/`).

As a result, the JSX code defined via `element` is rendered in place of the `<RouterProvider>` component by React Router. In this case, this means that the content of the `Dashboard` component is displayed because the `element` property value of this route definition is `<Dashboard />`. It is quite common to use single components (such as `<Dashboard />`, in this example), but you could set any JSX content as a value for the `element` property.

In the preceding example, no complex page is displayed. Instead, only some text shows up on the screen. This will change later in this chapter, though.

But it gets interesting if you manually change the URL from just `localhost:5173` to `localhost:5173/orders` in the browser address bar. In any of the previous chapters, this would not have changed the page content. But now, with routing enabled and the appropriate routes being defined, the page content does change, as shown:



The "Orders" route component

This component could display a list of orders.

It's also just a dummy example.

It's worth noting, that it's a regular React component that's activated by React Router because of the active route configuration.

Figure 13.2: For /orders, the content of the Orders component is displayed

Once the URL changes, the content of the `Orders` component is displayed on the screen. It's again just some basic text in this first example, but it shows that different code is rendered for different URL paths.

However, this basic example has a major flaw (besides the quite boring page content). Right now, users must enter URLs manually. But, of course, that's not how you typically use websites.

Adding Page Navigation

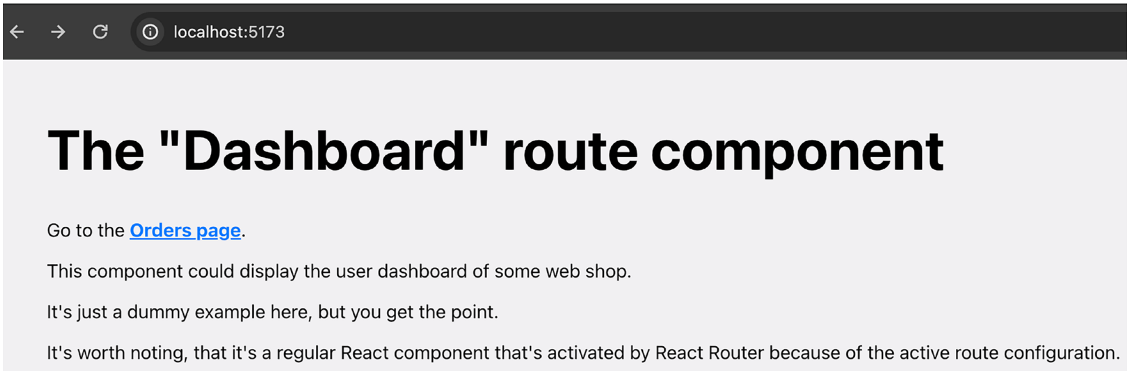
To allow users to switch between different website pages without editing the browser address bar manually, websites normally contain links, typically added via the `<a>` HTML element (the anchor element), like this:

```
<a href="/orders">Past Orders</a>
```

For this example, on-page navigation could therefore be added by modifying the `Dashboard` component code like this:

```
function Dashboard() {
  return (
    <>
      <h1>The "Dashboard" route component</h1>
      <p>Go to the <a href="/orders">Orders page</a>.</p>
      {/* <p> elements omitted */}
    </>
  );
}

export default Dashboard;
```

In this code snippet, a link to the `/orders` route has been added. Website visitors therefore see this page now:



The "Dashboard" route component

Go to the Orders page.

This component could display the user dashboard of some web shop.

It's just a dummy example here, but you get the point.

It's worth noting, that it's a regular React component that's activated by React Router because of the active route configuration.

Figure 13.3: A navigation link was added

When website users click this link, they are therefore taken to the `/orders` route and the content of the `Orders` component is displayed on the screen.

This approach works but has a major flaw: the website is reloaded every time a user clicks the link. You can tell that it's reloaded because the browser's refresh icon changes to a cross (briefly) whenever you click a link.

This happens because the browser sends a new HTTP request to the server whenever a link is clicked. Even though the server always returns the same single HTML page, the page is reloaded during that process (because of the new HTTP request that was sent).

While that's not a problem on this simple demo page, it would be an issue if you had some shared state (e.g., app-wide state managed via context) that should not be reset during a page change. In addition, every new request takes time and forces the browser to download all website assets (e.g., script files) again. Even though those files might be cached, this is an unnecessary step that may impact website performance.

The following, slightly adjusted, example `App` component illustrates the state-resetting problem:

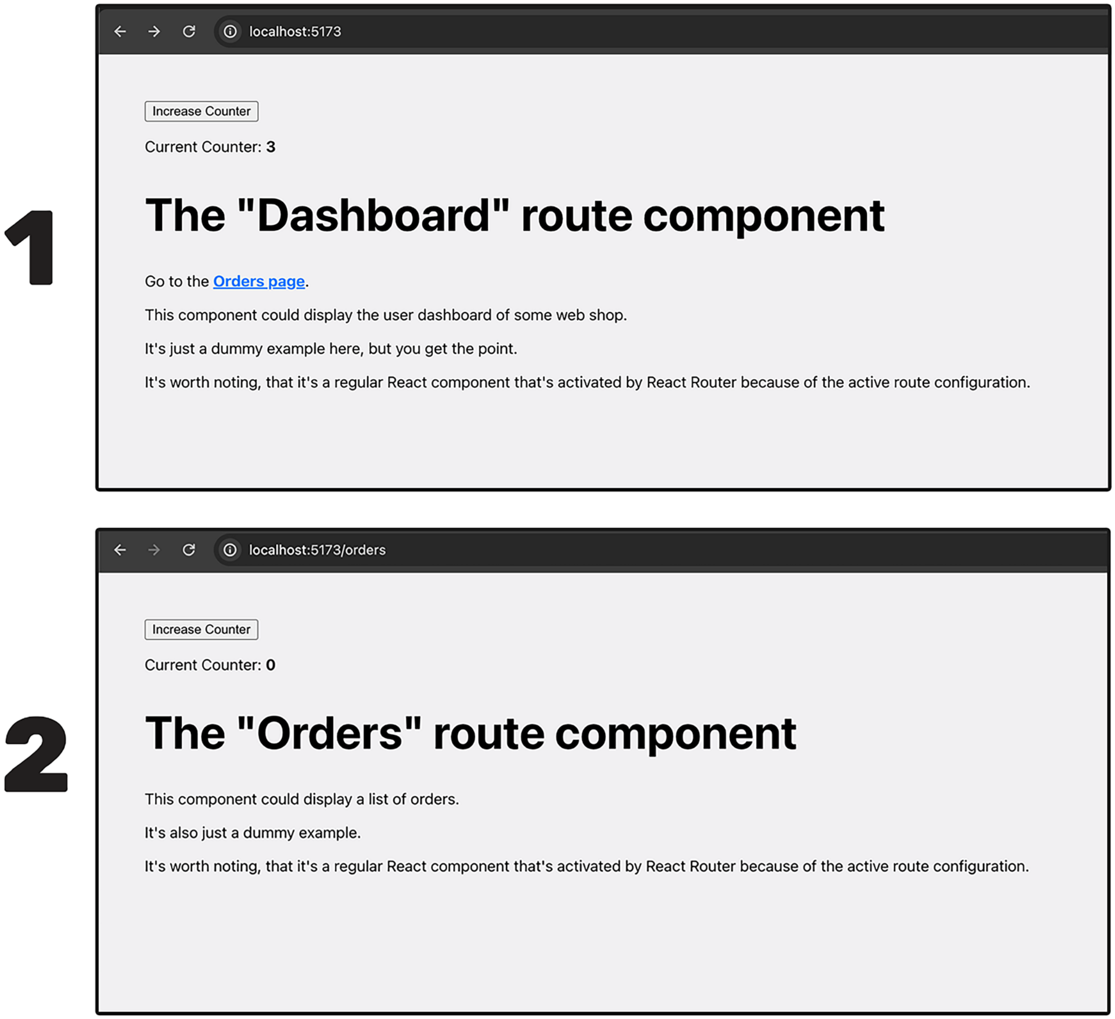
```
import { useState } from 'react';
import {
  createBrowserRouter,
  RouterProvider
} from 'react-router-dom';

import Dashboard from './routes/Dashboard.jsx';
import Orders from './routes/Orders.jsx';

const router = createBrowserRouter([
  { path: '/', element: <Dashboard /> },
  { path: '/orders', element: <Orders /> },
]);

function App() {
  const [counter, setCounter] = useState(0);

  function handleIncCounter() {
    setCounter((prevCounter) => prevCounter + 1);
  }

  return (
    <>
      <p>
        <button onClick={handleIncCounter}>Increase Counter</button>
      </p>
      <p>Current Counter: <strong>{counter}</strong></p>
      <RouterProvider router={router} />
    </>
  );
}

export default App;
```

Note

The code for this example can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/examples/03-naive-navigation-problem`.

In this example, a simple counter was added to the `App` component. Since `<RouterProvider>` is rendered in that same component, below the counter, the `App` component should not be replaced when a user visits a different page (instead, it's `<RouterProvider>` that should be replaced—not the entire `App` component JSX code).

At least, that's the theory. But, as you can see in the following screenshot, the `counter` state is lost whenever any link is clicked:



Current Counter: **3**

The "Dashboard" route component

Go to the Orders page.

This component could display the user dashboard of some web shop.

It's just a dummy example here, but you get the point.

It's worth noting, that it's a regular React component that's activated by React Router because of the active route configuration.

Current Counter: **0**

The "Orders" route component

This component could display a list of orders.

It's also just a dummy example.

It's worth noting, that it's a regular React component that's activated by React Router because of the active route configuration.

Figure 13.4: The counter state is reset when switching the page

In the screenshot, you can see that the counter is initially set to `3` (because the button was clicked thrice). After navigating from `Dashboard` to the `Orders` page (via clicking the `Orders page` link), the counter changes to `0`.

That happens because the page is reloaded due to the HTTP request that's sent by the browser.

To work around this issue and avoid this unintended page reload, you must prevent the browser's default behavior. Instead of sending a new HTTP request, the browser URL address should just be updated (from `localhost:5173` to `localhost:5173/orders`) and the target component (`Orders`) should be loaded. Therefore, to the website user, it would seem as if a different page was loaded. But behind the scenes, it's just the page document (the DOM) that was updated.

Thankfully, you don't have to implement the logic for this on your own. Instead, the React Router library exposes a special `Link` component that should be used instead of the anchor `<a>` element.

To use this new component, the code in `src/routes/Dashboard.jsx` must be adjusted like this:

```
import { Link } from 'react-router-dom';

function Dashboard() {
  return (
    <>
      <h1>The "Dashboard" route component</h1>
      <p>Go to the <Link to="/orders">Orders page</Link>.</p>
      <p>
        This component could display the user dashboard
        of some web shop.
      </p>
      <p>It's just a dummy example here, but you get the point.</p>
      <p>
        It's worth noting, that it's a regular React component
        that's activated by React Router because of the
        active route configuration.
      </p>
    </>
  );
}

export default Dashboard;
```

Note

The code for this example can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/examples/04-react-router-navigation`.

Inside this updated example, the new `Link` component is used. That component requires a `to` prop, which is used to define the URL path that should be loaded.

By using this component in place of the `<a>` anchor element, the counter state is no longer reset. This is because React Router now prevents the browser's default behavior (i.e., the unintended page reload described above) and displays the correct page content.

Under the hood, the `Link` component still renders the built-in `<a>` element. But React Router controls it and implements the behavior described above.

The `Link` component is therefore the default component that should be used for internal links. For external links, the standard `<a>` element should be used instead since the link leads away from the website, hence there is no state to preserve or page reload to prevent.

Working with Layouts & Nested Routes

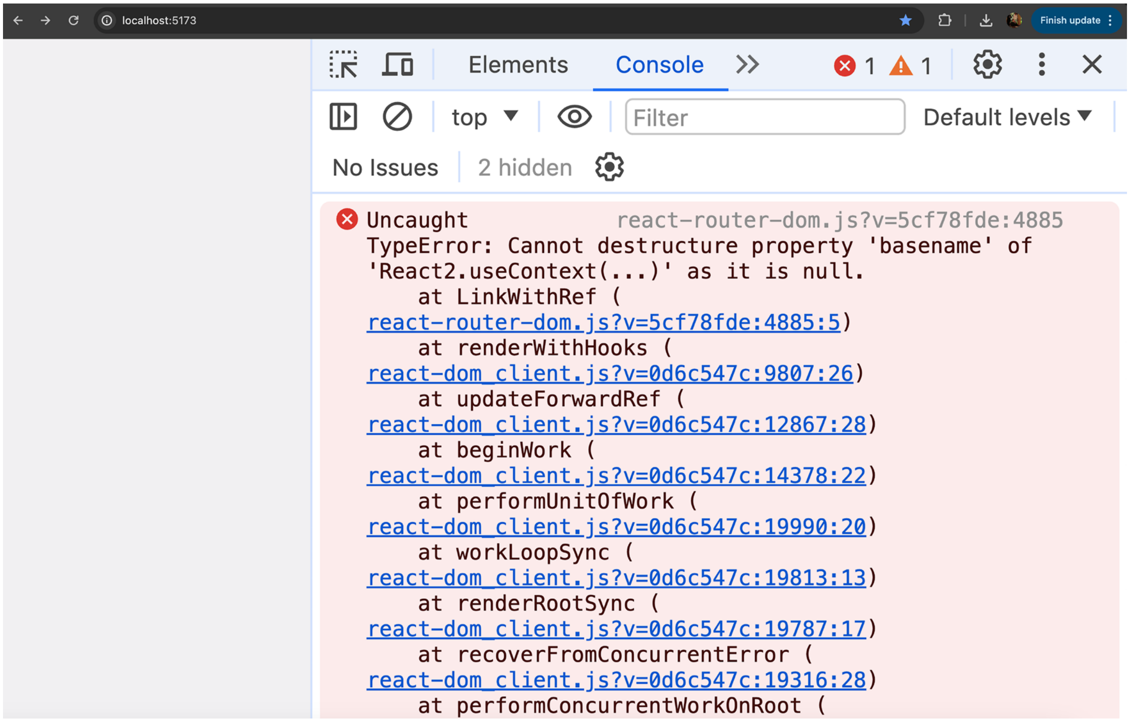
Most websites require some form of page-wide navigation (and hence navigation links) or other page sections that should be shared across some or all routes.

Consider the previous example website with the routes `/` and `/orders`. The example website would also benefit from having a top navigation bar that allows users to switch between the starting page (i.e., the `Dashboard` route) and the `Orders` page.

Therefore, `App.jsx` could be adjusted to have a top navigation bar inside a `<header>` above `<RouterProvider>`:

```
import {
  createBrowserRouter,
  RouterProvider,
  Link
} from 'react-router-dom';

import Dashboard from './routes/Dashboard.jsx';
import Orders from './routes/Orders.jsx';

const router = createBrowserRouter([
  { path: '/', element: <Dashboard /> },
  { path: '/orders', element: <Orders /> },
]);

function App() {
  return (
    <>
      <header>
        <nav>
          <ul>
            <li>
              <Link to="/">My Dashboard</Link>
            </li>
            <li>
              <Link to="/orders">Past Orders</Link>
            </li>
          </ul>
        </nav>
      </header>
      <RouterProvider router={router} />
    </>
  );
}

export default App;
```

But if you try to run this application, you'll see a blank page and encounter an error message in the JavaScript console in the browser developer tools.



```
Uncaught          react-router-dom.js?v=5cf78fde:4885
TypeError: Cannot destructure property 'basename' of
'React2.useContext(...)' as it is null.
    at LinkWithRef (
react-router-dom.js?v=5cf78fde:4885:5)
    at renderWithHooks (
react-dom_client.js?v=0d6c547c:9807:26)
    at updateForwardRef (
react-dom_client.js?v=0d6c547c:12867:28)
    at beginWork (
react-dom_client.js?v=0d6c547c:14378:22)
    at performUnitOfWork (
react-dom_client.js?v=0d6c547c:19990:20)
    at workLoopSync (
react-dom_client.js?v=0d6c547c:19813:13)
    at renderRootSync (
react-dom_client.js?v=0d6c547c:19787:17)
    at recoverFromConcurrentError (
react-dom_client.js?v=0d6c547c:19316:28)
    at performConcurrentWorkOnRoot (
```

Figure 13.5: React Router seems to complain about something

The error message is a bit cryptic, but the problem is that the above code tries to use `<Link>` outside of a component controlled by React Router.

Only components loaded via `<RouterProvider>` are controlled by React Router, hence React Router features like its `Link` component can only be used in route components (or their descendent components).

Therefore, setting up the main navigation inside of the `App` component (which is **not** loaded by React Router) does not work.

To wrap or enhance multiple route components with some shared component and JSX markup, you must define a new route that wraps the existing routes. Such a route is also sometimes called a **layout route** since it can be used to provide some shared layout. The routes wrapped by this route would be called **nested routes**.

A layout route is defined like any other route inside the route definitions array. It then becomes a layout route by wrapping other routes via a special `children` property that's accepted by React Router. That `children` property receives an array of nested routes—child routes to the wrapping parent route.

Here's the adjusted route definition code for this example app:

```
import Root from './routes/Root.jsx';
import Dashboard from './routes/Dashboard.jsx';
import Orders from './routes/Orders.jsx';

const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    children: [
      { index: true, element: <Dashboard /> },
      { path: '/orders', element: <Orders /> },
    ],
  },
]);
```

In this updated code snippet, a new root layout route is defined—a route that registers the existing routes (the `Dashboard` and `Orders` components) as child routes. This setup therefore allows the `Root` component to be active simultaneously to the `Dashboard` or `Orders` route component.

You might also note that the `Dashboard` route no longer has a `path`. Instead, it now has an `index` property, which is set to `true`. That `index` property is a property that can be used when working with nested routes. It tells React Router which nested route to activate (and therefore which component to load) if the parent route path is matched exactly.

In this example, when the `/` path is active (i.e., if a user visits `<domain>/`), the `Root` and `Dashboard` components will be rendered. For `<domain>/orders`, `Root` and `Orders` would become visible.

The `Root` component is a newly added component in this example. It's a standard component (like `Dashboard` or `Orders`) with one special feature: it defines the place where the child route components should be inserted via a special `Outlet` component that's provided by React Router:

```
import { Link, Outlet } from 'react-router-dom';

function Root() {
  return (
    <>
      <header>
        <nav>
          <ul>
```

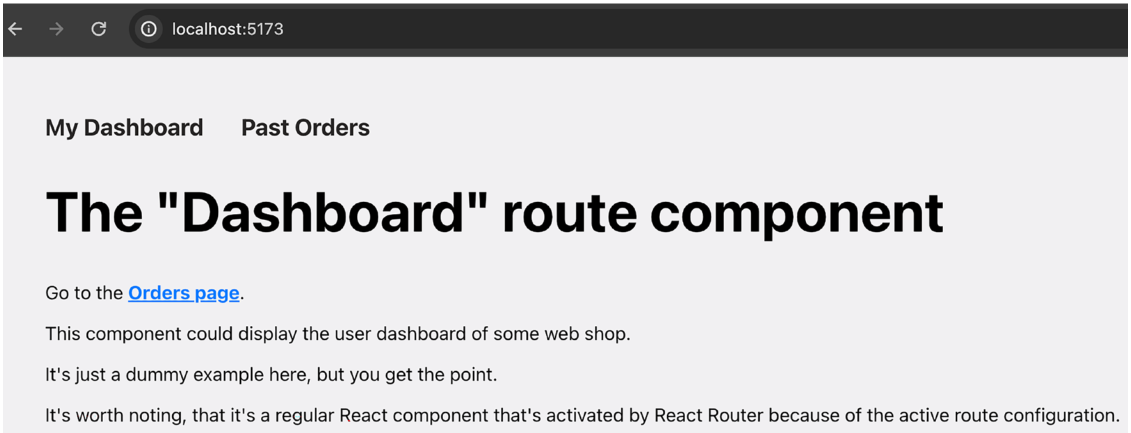
```
            <li>
              <Link to="/">My Dashboard</Link>
            </li>
            <li>
              <Link to="/orders">Past Orders</Link>
            </li>
          </ul>
        </nav>
      </header>
      <Outlet />
    </>
  );
}

export default Root;
```

The `<Outlet />` placeholder is needed since React Router must know where to render the route components of the routes passed to the `children` property.

Note

You can find the complete example code on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/examples/05-layouts-nested-routes`.

Since the `Root` component itself is also rendered by React Router, it now is a component that has access to the `<Link>` tag. Therefore, this `Root` component can be used to share common markup (like the navigation `<header>`) across all nested routes.



My Dashboard Past Orders

The "Dashboard" route component

Go to the Orders page.

This component could display the user dashboard of some web shop.

It's just a dummy example here, but you get the point.

It's worth noting, that it's a regular React component that's activated by React Router because of the active route configuration.

Figure 13.6: A shared navigation bar is displayed at the top (for all routes)

Hence, nested routes and layout routes (or wrapper routes) are crucial features offered by React Router.

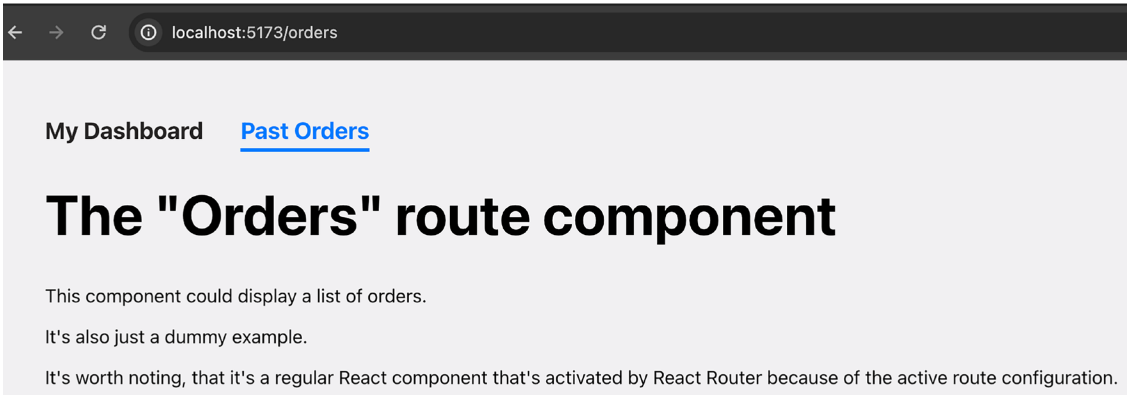
It's also worth noting that you can add as many levels of route nesting as needed by your application—you're **not** restricted to having just one layout route that wraps child routes.

From Link to NavLink

In a shared navigation, as set up in the previous chapter, you often want to highlight the link that led to the currently active page. For example, if a user clicked the `Past Orders` link (and hence navigates to `/orders`), that link should change its appearance (e.g., its color).

Consider the example from previously (*Figure 13.6*)—there, in the top navigation bar, it's not immediately obvious whether the user is on the `Dashboard` page or the `Orders` page. Of course, the URL address and the main page content do change, but the navigation items don't adjust visually.

To prove this point, compare the previous screenshot to the following one:



The "Orders" route component

This component could display a list of orders.

It's also just a dummy example.

It's worth noting, that it's a regular React component that's activated by React Router because of the active route configuration.

Figure 13.7: The highlighted "Past Orders" navigation link is underlined and changes its color

In this version of the website, it's immediately clear that the user is on the `"Orders"` page since the `Past Orders` navigation link is highlighted. It's subtle things such as this that make websites more usable and can ultimately lead to higher user engagement.

But how can this be achieved?

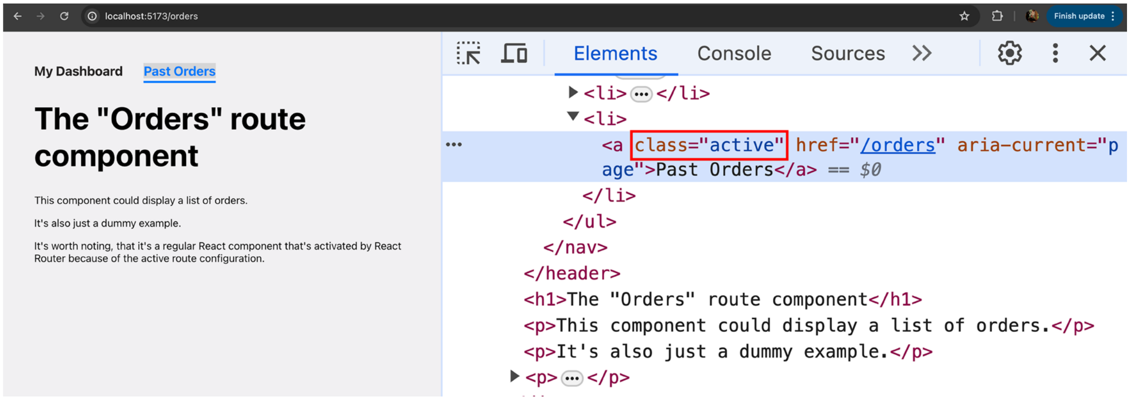
To do this, you would not use the `Link` component, but instead, a special alternative component offered by `react-router-dom`: the `NavLink` component:

```
import { NavLink, Outlet } from 'react-router-dom';

function Root() {
  return (
    <>
      <header>
        <nav>
          <ul>
```

```
            <li>
              <NavLink to="/">My Dashboard</NavLink>
            </li>
            <li>
              <NavLink to="/orders">Past Orders</NavLink>
            </li>
          </ul>
        </nav>
      </header>
      <Outlet />
    </>
  );
}

export default Root;
```

The `NavLink` component is used pretty much like the `Link` component. You wrap it around some text (the link's caption), and you define the target path via the `to` prop. However, the `NavLink` component has some extra styling-related features the regular `Link` component does not have.

To be precise, the `NavLink` component by default applies a CSS class called `active` to the rendered anchor element when the link is active.



My Dashboard Past Orders

The "Orders" route component

This component could display a list of orders.

It's also just a dummy example.

It's worth noting, that it's a regular React component that's activated by React Router because of the active route configuration.

```
▶<li>⋯</li>
▼<li>
   <a class="active" href="/orders" aria-current="page">Past Orders</a> == $0
 </li>
</ul>
</nav>
</header>
<h1>The "Orders" route component</h1>
<p>This component could display a list of orders.</p>
<p>It's also just a dummy example.</p>
▶<p>⋯</p>
```

Figure 13.8: The rendered <a> element received an "active" CSS class

In case you want to apply a different CSS class name or inline styles when a link becomes active, `NavLink` also allows you to do that. Because `NavLink`'s `className` and `style` props behave slightly differently than they do on other elements. Besides accepting string values (`className`) or style objects (`style`), both props also accept functions that will automatically be called by React Router upon every navigation action. For example, the following code could be used to ensure that a certain CSS class or style is applied:

```
<NavLink
```

```
  className={({ isActive }) => isActive ? 'loaded' : ''}
  style={({ isActive }) => isActive ? { color: 'red' } : undefined}>
    Some Link
</NavLink>
```

In the above code snippet, both `className` and `style` take advantage of the function that will be executed by React Router. This function automatically receives an object as an input argument—an object that's created and provided by React Router, and that contains an `isActive` property. React Router sets `isActive` to `true` if the link leads to the currently active route, and to `false` otherwise.

You can therefore return any CSS class names or style objects of your choosing in those functions. React Router will then apply them to the rendered `<a>` element.

> **Note**
>
> You can find the finished code for this example on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/examples/06-navlinks`.

One important note is that `NavLink` will consider a route to be active if its path matches the current URL path *or* if its path starts with the current URL path. For example, if you had a `/blog/all-posts` route, a `NavLink` component that points at just `/blog` would be considered active if the current route is `/blog/all-posts` (because that route path starts with `/blog`). If you don't want this behavior, you can add the special `end` prop to the `NavLink` component, as follows:

```
<NavLink
  to="/blog"
  style={({ isActive }) => isActive ? { color: 'red' } : undefined}
  end>
    Blog
</NavLink>
```

With this special prop added, this `NavLink` would only be considered active if the current route is exactly `/blog`—for `/blog/all-posts`, the link would not be active.

An exception from that rule would be links to just `/`. Since all routes technically start with this "empty path," React Router by default only considers `<NavLink to="/">` as active if the user is currently on `<domain>/`. For other paths (e.g., `/orders`), `<NavLink to="/">` would not be marked as active.

`NavLink` is always the preferred choice when the styling of a link depends on the currently active route. For all other internal links, use `Link`. For external links, `<a>` is the element of choice.

Route Components versus "Normal" Components

It's worth mentioning and noting that, in the previous examples, the `Dashboard` and `Orders` components were regular React components. You could use these components anywhere in your React app—not just as values for the `element` property of a route definition.

However, the two components are special in that both are stored in the `src/routes` folder in the project directory. They are not stored in the `src/components` folder, which was used for components throughout this book.

That's not something you have to do, though. Indeed, the folder names are entirely up to you. These two components could be stored in `src/components`. You could also store them in an `src/elements` folder. But using `src/routes` is quite common for components that are exclusively used for routing. Popular alternatives are `src/screens`, `src/views`, and `src/pages` (again, it is up to you).

If your app includes any other components that are not used as routing elements, you would still store those in `src/components` (i.e., in a different path). This is not a hard rule or a technical requirement, but it does help with keeping your React projects manageable. Splitting your components across multiple folders makes it easier to quickly understand which components fulfill which purposes in the project.

In the example project mentioned previously, you can, for example, refactor the code such that the navigation code is stored in a separate component (e.g., a `MainNavigation` component, stored in `src/components/shared/MainNavigation.jsx`). The component file code looks like this:

```
import { NavLink } from 'react-router-dom';

import classes from './MainNavigation.module.css';

function MainNavigation() {
  return (
    <header className={classes.header}>
      <nav>
        <ul>
          <li>
            <NavLink
              to="/"
              className={({ isActive }) =>
                isActive ? classes.active : undefined
              }
              end
            >
              My Dashboard
            </NavLink>
          </li>
          <li>
            <NavLink
              to="/orders"
              className={({ isActive }) =>
                isActive ? classes.active : undefined
              }
            >
              Past Orders
            </NavLink>
          </li>
        </ul>
      </nav>
    </header>
  );
}

export default MainNavigation;
```

In this code snippet, the `NavLink` component is adjusted to assign a CSS class named `active` to any link that belongs to the currently active route. This is required when using CSS Modules since the class names are changed during the build process, as discussed in *Chapter 6*, *Styling React Apps*. Besides that, it's essentially the same navigation menu code as that used earlier in this chapter.

This `MainNavigation` component can then be imported and used in the `Root.jsx` file like this:

```
import { Outlet } from 'react-router-dom';

import MainNavigation from '../components/shared/MainNavigation.jsx';

function Root() {
  return (
    <>
      <MainNavigation />
      <Outlet />
    </>
  );
}

export default Root;
```

Importing and using the `MainNavigation` component leads to a leaner `Root` component and yet preserves the same functionality as before.

These changes show how you can combine routing components that are only used for routing (`Dashboard` and `Orders`) and components that are used outside of routing (`MainNavigation`).

Note

You can find the finished code for this example on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/examples/07-routing-and-normal-cmp`.

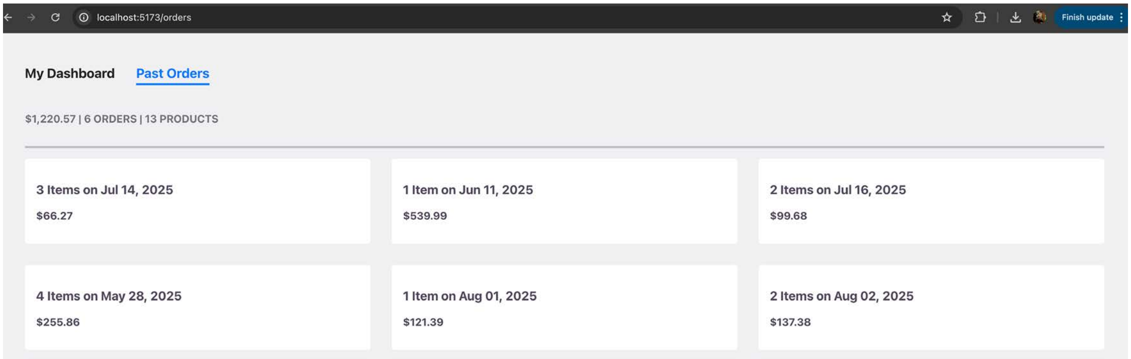
But even with those markup and style improvements, the demo application still suffers from an important problem: it only supports static, predefined routes. But, for most websites, those kinds of routes are not enough.

From Static to Dynamic Routes

Thus far, all examples have had two routes: `/` for the `Dashboard` component and `/orders` for the `Orders` component. But you can, of course, add as many routes as needed. If your website consists of 20 different pages, you can (and should) add 20 route definitions (i.e., 20 `Route` components) to your `App` component.

On most websites, however, you will also have some routes that can't be defined manually—because not all routes and their exact paths are known in advance.

Consider the example from before, enriched with additional components and some dummy data:



$1,220.57 | 6 ORDERS | 13 PRODUCTS

3 Items on Jul 14, 2025
$66.27

1 Item on Jun 11, 2025
$539.99

2 Items on Jul 16, 2025
$99.68

4 Items on May 28, 2025
$255.86

1 Item on Aug 01, 2025
$121.39

2 Items on Aug 02, 2025
$137.38

Figure 13.9: A list of order items

> **Note**
>
> You can find the code for this example on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/examples/08-dynamic-routes-problem`. In the code, you'll notice that many new components and style files were added. The code does not use any new features, though. It's just used to display a more realistic user interface and output some dummy data.

In the preceding screenshot, *Figure 13.9*, you can see a list of order items being output on the `Past Orders` page (i.e., by the `Orders` component).

In the underlying code, every order item is wrapped with a `Link` component so that a separate page with more details can be loaded for each item:

```
function OrdersList() {
  return (
    <ul className={classes.list}>
      {orders.map((order) => (
          <li key={order.id}>
            <Link to='/orders'><OrderItem order={order} /></Link>
          </li>
        ))}
      </ul>
    );
  }
```

In this code snippet, the path for the `Link` component is set to `/orders`. However, that's not the final value that should be assigned. Instead, this example highlights an important problem: while it's the same route and component that should be loaded for every order item (i.e., some component that displays detailed data about the selected order), the exact content output by that component depends on which order item was selected. It's the same route and component with different data.

Outside of routing, you would use props to reuse the same component with different data. But with routing, it's not just about the component. You also must support different paths—because the detailed data for different orders should be loaded via different paths (e.g., `/orders/o1`, `/orders/o2`, etc.). Otherwise, you would again end up with URLs that are not shareable or reloadable.

Therefore, the path must include not only some static identifier (such as `/orders`) but also a dynamic value that's different for every order item. For three order items with `id` values `o1`, `o2`, and `o3`, the goal could be to support the `/orders/o1`, `/orders/o2`, and `/order/o3` paths.

For this reason, the following three route definitions could be added:

```
{ path: '/orders/o1', element: <OrderDetail id="o1" /> },
{ path: '/orders/o2', element: <OrderDetail id="o2" /> },
{ path: '/orders/o3', element: <OrderDetail id="o3" /> }
```

But this solution has a major flaw. Adding all these routes manually is a huge amount of work. And that's not even the biggest problem. You typically don't even know all values in advance. In this example, when a new order is placed, a new route would have to be added. But you can't adjust the source code of your website every time a visitor places an order.

Clearly, then, a better solution is needed. React Router offers that better solution as it supports **dynamic routes**.

Dynamic routes are defined just like other routes, except that, when defining their `path` values, you will need to include one or more **dynamic path segments** with identifiers of your choice.

The `OrderDetail` route definition therefore looks like this:

```
{ path: '/orders/:id', element: <OrderDetail /> }
```

The following three key things have changed:

- It's just one route definition instead of a (possibly) infinite list of definitions.
- `path` contains a dynamic path segment (`:id`).
- `OrderDetail` no longer receives an `id` prop.

The `:id` syntax is a special syntax supported by React Router. Whenever a segment of a path starts with a colon, React Router treats it as a **dynamic segment**. That means that it will be replaced with a different value in the actual URL path. For the `/orders/:id` route path, the `/orders/o1`, `/orders/o2`, and `/orders/abc` paths would all match and therefore activate the route.

Of course, you don't have to use `:id`. You can use any identifier of your choice. For the preceding example, `:orderId`, `:order`, or `:oid` would also make sense.

The identifier will help your app access the correct data inside the page component that should be loaded for the dynamic route (i.e., the `OrderDetail` route component in the example code snippets above). That's why the `id` prop was removed from `OrderDetail` in the last code snippet. Since only one route is defined, only one specific `id` value could be passed via props. That won't help. Therefore, a different way of loading order-specific data must be used.

Extracting Route Parameters

In the previous example, when a website user visits `/orders/o1` or `/orders/o2` (or the same path for any other order ID), the `OrderDetail` component is loaded. This component should then output more information about the specific order that was selected (i.e., the order whose ID is encoded in the URL path).

By the way, that's not just the case for this example; you can think of many other types of websites as well. You could also have, for example, an online shop with routes for products (`/products/p1`, `/products/p2`, etc.), or a travel blog where users can visit individual blog posts (`/blog/post1`, `/blog/post2`, etc.).

In all these cases, the question is how do you get access to the data that should be loaded for the specific identifier (e.g., the ID) that's included in the URL path? Since it's always the same component that's loaded, you need a way of dynamically identifying the order, product, or blog post for which the detail data should be fetched.

One possible solution would be the usage of props. Whenever you build a component that should be reusable yet configurable and dynamic, you can use props to accept different values. For example, the `OrderDetail` component could accept an `id` prop and then, inside the component function body, load the data for that specific order ID.

However, as mentioned in the previous section, this is not a possible solution when loading the component via routing. Keep in mind that the `OrderDetail` component is created when defining the route:

```
{ path: '/orders/:id', element: <OrderDetail /> }
```

Since the component is created when defining the route in the `App` component, you can't pass in any dynamic, ID-specific prop values.

Fortunately, though, that's not necessary. React Router gives you a solution that allows you to extract the data encoded in the URL path from inside the component that's displayed on the screen (when the route becomes active): the `useParams()` Hook.

This Hook can be used to get access to the route parameters of the currently active route. Route parameters are simply the dynamic values encoded in the URL path—`id`, in the case of this `OrderDetail` example.

Inside the `OrderDetail` component, `useParams()` can therefore be used to extract the specific order ID and load the appropriate order data, as follows:

```
import { useParams } from 'react-router-dom';

import Details from '../components/orders/Details.jsx';
import { getOrderById } from '../data/orders.js';

function OrderDetail() {
  const params = useParams();
  const orderId = params.id; // orderId is "o1", "o2" etc.
  const order = getOrderById(orderId);

  return <Details order={order} />;
}

export default OrderDetail;
```

As you can see in this snippet, `useParams()` returns an object that contains all route parameters of the currently active route as properties. Since the route path was defined as `/orders/:id`, the `params` object contains an `id` property. The value of that property is then the actual value encoded in the URL path (e.g., `o1`). If you choose a different identifier name in the route definition (e.g., `/orders/:orderId` instead of `/orders/:id`), that property name must be used to access the value in the `params` object (i.e., access `params.orderId`).

Note

You can find the complete code on GitHub at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/examples/09-dynamic-routes`.

By using route parameters, you can thus easily create dynamic routes that lead to different data being loaded. But, of course, defining routes and handling route activation are not that helpful if you do not have links leading to dynamic routes.

Creating Dynamic Links

As mentioned earlier in this chapter (in the *Adding Page Navigation* section), website visitors should be able to click on links that should then take them to the different pages that make up the overall website—meaning, those links should activate the various routes defined with the help of React Router.

As explained in the *Adding Page Navigation* and *From Link to NavLink* sections, for internal links (i.e., links leading to routes defined inside the React app), the `Link` or `NavLink` components are used.

So, for static routes such as `/orders`, links are created like this:

```
<Link to="/orders">Past Orders</Link> // or use <NavLink> instead
```

When building a link to a dynamic route such as `/orders/:id`, you can therefore simply create a link like this:

```
<Link to="/orders/o1">Past Orders</Link>
```

This specific link loads the `OrderDetails` component for the order with the ID `o1`.

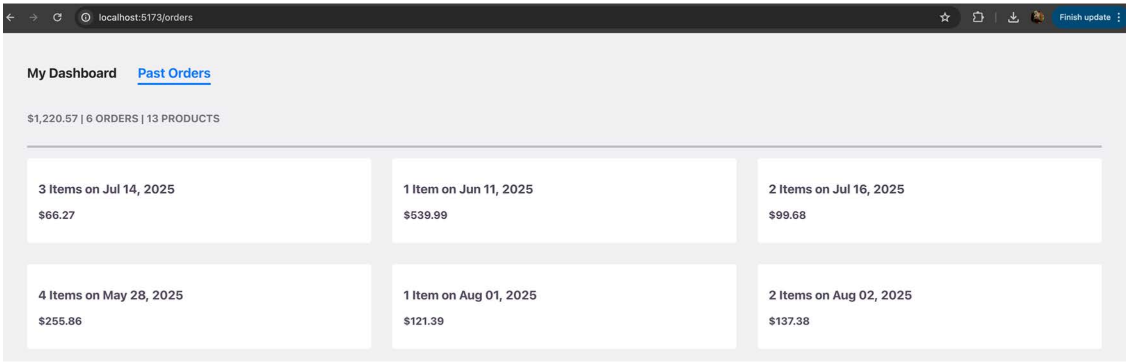
Building the link as follows would be incorrect:

```
<Link to="/orders/:id">Past Orders</Link>
```

The dynamic path segment syntax (`:id`) is only used when defining the route—not when creating a link. The link has to lead to a specific resource (a specific order, in this case).

However, creating links to specific orders, as shown previously, is not very practical. Just as it wouldn't make sense to define all dynamic routes individually (see the *From Static to Dynamic Routes* section), it doesn't make sense to create the respective links manually.

Sticking to the orders example, there is also no need to create links like that as you already have a list of orders that's output on one page (the `Orders` component, in this case). Similarly, you could have a list of products in an online shop. In all these cases, the individual items (orders, products, etc.) should be clickable and lead to details pages with more information.



My Dashboard Past Orders

$1,220.57 | 6 ORDERS | 13 PRODUCTS

3 Items on Jul 14, 2025
$66.27

1 Item on Jun 11, 2025
$539.99

2 Items on Jul 16, 2025
$99.68

4 Items on May 28, 2025
$255.86

1 Item on Aug 01, 2025
$121.39

2 Items on Aug 02, 2025
$137.38

Figure 13.10: A list of clickable order items

Therefore, the links can be generated dynamically when rendering the list of JSX elements. In the case of the orders example, the code looks like this:

```
function OrdersList() {
  return (
    <ul className={classes.list}>
      {orders.map((order) => (
          <li key={order.id}>
            <Link
              to={`/orders/${order.id}`}>
              <OrderItem order={order} />
            </Link>
          </li>
        ))}
      </ul>
    );
  }
```

In this code example, the value of the `to` prop is set dynamically equal to a string that includes the `order.id` value. Therefore, every list item receives a unique link that leads to a different details page. Or, to be precise, the link always leads to the same component but with a different order `id` value, hence loading different order data.

Note

In this code snippet (which can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/examples/10-dynamic-links`), the string is created as a **template literal**. That's a default JavaScript feature that simplifies the creation of strings that include dynamic values.

You can learn more about template literals on MDN at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals`.

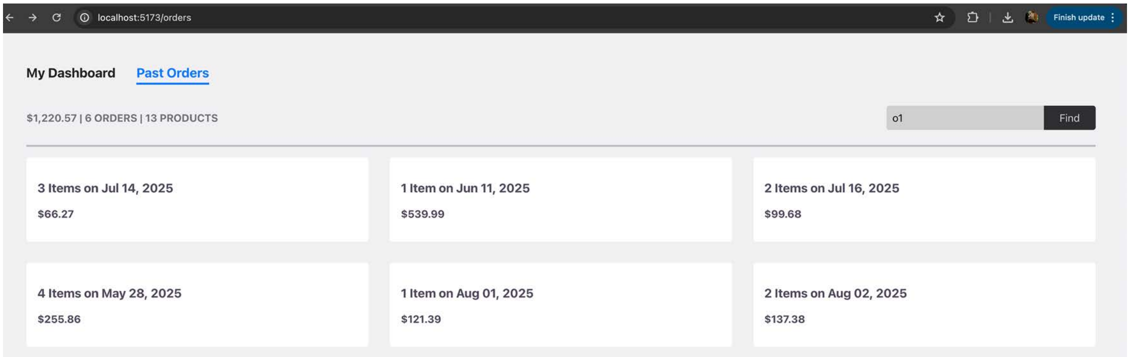
Navigating Programmatically

In the previous section, as well as earlier in this chapter, user navigation was enabled by adding links to the website. Indeed, links are the default way of adding navigation to a website. But there are scenarios where programmatic navigation is required instead.

Programmatic navigation means that a new page should be loaded via JavaScript code (rather than using a link). This kind of navigation is typically required if the active page changes in response to some action—e.g., upon form submission.

If you take the example of form submission, you will normally want to extract and save the submitted data. But thereafter, the user will sometimes need to be redirected to a different page. For example, it makes no sense to keep the user on a `Checkout` page after processing the entered credit card details. You might want to redirect the user to a `Success` page instead.

In the example discussed throughout this chapter, the `Past Orders` page could include an input field that allows users to directly enter an order ID and load the respective order data after clicking the `Find` button.


localhost:5173/orders

My Dashboard Past Orders

$1,220.57 | 6 ORDERS | 13 PRODUCTS

o1 Find

3 Items on Jul 14, 2025
$66.27

1 Item on Jun 11, 2025
$539.99

2 Items on Jul 16, 2025
$99.68

4 Items on May 28, 2025
$255.86

1 Item on Aug 01, 2025
$121.39

2 Items on Aug 02, 2025
$137.38

Figure 13.11: An input field that can be used to quickly load a specific order

In this example, the entered order ID is first processed and validated before the user is sent to the respective details page. If the provided ID is invalid, an error message is shown instead. The code looks like this:

```
import orders, { getOrdersSummaryData } from '../../data/orders.js';
import classes from './OrdersSummary.module.css';

function OrdersSummary() {
  const { quantity, total } = getOrdersSummaryData();

  const formattedTotal = new Intl.NumberFormat('en-US', {
    style: 'currency',
    currency: 'USD',
  }).format(total);

  function findOrderAction(formData) {
    const orderId = formData.get('order-id');
    const orderExists = orders.some((order) => order.id === orderId);

    if (!orderExists) {
      alert('Could not find an order for the entered id.');
      return;
    }
  }

  return (
    <div className={classes.row}>
      <p className={classes.summary}>
        {formattedTotal} | {orders.length} Orders |
          {quantity} Products
      </p>
      <form className={classes.form} action={findOrderAction}>
        <input
          type="text"
          name="order-id"
          placeholder="Enter order id"
          aria-label="Find an order by id."
        />
        <button>Find</button>
      </form>
    </div>
  );
}

export default OrdersSummary;
```

The code snippet does not yet include the code that will actually trigger the page change, but it does show how the user input is read and validated.

Therefore, this is a perfect scenario for the use of programmatic navigation. A link can't be used here since it would immediately trigger a page change—without allowing you to validate the user input first (at least not after the link was clicked).

The React Router library also supports programmatic navigation for cases like this. You can import and use the special `useNavigate()` Hook to gain access to a navigation function that can be used to trigger a navigation action (i.e., a page change):

```
import { useNavigate } from 'react-router-dom';

const navigate = useNavigate();
navigate('/orders');
// programmatic alternative to <Link to="/orders">
```

Hence, the `OrdersSummary` component from previously can be adjusted like this to use this new Hook:

```
function OrdersSummary() {
  const navigate = useNavigate();

  const { quantity, total } = getOrdersSummaryData();

  const formattedTotal = new Intl.NumberFormat('en-US', {
    style: 'currency',
    currency: 'USD',
  }).format(total);
  function findOrderAction(formData) {
    const orderId = formData.get('order-id');
    const orderExists = orders.some((order) => order.id === orderId);

    if (!orderExists) {
      alert('Could not find an order for the entered id.');
      return;
    }

    navigate(`/orders/${orderId}`);
  }
  // returned JSX code did not change, hence omitted
}
```

It's worth noting that the value passed to `navigate()` is a dynamically constructed string. Programmatic navigation supports both static and dynamic paths.

Note

The code for this example can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/examples/11-programmatic-navigation`.

Redirecting

Thus far, all the explored navigation options (links and programmatic navigation) forward a user to a specific page.

In most cases, that's the intended behavior. But in some cases, the goal is to redirect a user instead of forwarding them.

The difference is subtle but important. When a user is forwarded, they can use the browser's navigation buttons (`Back` and `Forward`) to go back to the previous page or forward to the page they came from. For redirects, that's not possible. Whenever a user is redirected to a specific page (rather than forwarded), they can't use the `Back` button to return to the previous page.

Redirecting users can, for example, be useful for ensuring that users can't go back to a login page after authenticating successfully.

When using React Router, the default behavior is to forward users. But you can easily switch to redirecting by adding the special `replace` prop to the `Link` (or `NavLink`) components, as follows:

```
<Link to="/success" replace>Confirm Checkout</Link>
```

When using programmatic navigation, you can pass a second, optional argument to the `navigate()` function. That second parameter value must be an object that can contain a `replace` property that should be set to `true` if you want to redirect users:

```
navigate('/dashboard', { replace: true });
```

Being able to redirect or forward users allows you to build highly user-friendly web applications that offer the best possible user experience for different scenarios.

Handling Undefined Routes

Previous sections in this chapter have all assumed that you have predefined routes that should be reachable by website visitors. But what if a visitor enters a URL that's simply not supported?

For example, the demo website used throughout this chapter supports the `/`, `/orders`, and `/orders/<some-id>` paths. But it does not support `/home`, `/products/p1`, `/abc`, or any other path that's not one of the defined route paths.

To show a custom *Not Found* page, you can define a "catch all" route with a special path—the `*` path:

```
{ path: '*', element: <NotFound /> }
```

When adding this route to the list of route definitions in the `App` component, the `NotFound` component will be displayed on the screen when no other route matches the entered or generated URL path.

Lazy Loading

In *Chapter 10, Behind the Scenes of React and Optimization Opportunities*, you learned about lazy loading—a technique that can be used to load certain pieces of the React application code only when needed.

Code splitting makes a lot of sense if some components will be loaded conditionally and may not be needed at all. Hence, routing is a perfect scenario for lazy loading. When applications have multiple routes, some routes may never be visited by a user. Even if all routes are visited, not all the code for all app routes (i.e., for their components) must be downloaded right at the start when the application loads. Instead, it makes sense to only download code for individual routes when they actually become active.

Thankfully, React Router has built-in support for lazy loading and route-based code splitting. It provides a `lazy` property that can be added to a route definition. That property expects a function that dynamically imports the lazily loaded file (which contains the component that should be rendered). React Router then takes care of the rest—for example, you don't need to wrap `Suspense` around any components:

```
import {
  createBrowserRouter,
  RouterProvider
} from 'react-router-dom';

import Root from './routes/Root.jsx';
import Dashboard from './routes/Dashboard.jsx';
```

```
// Removed static imports of Orders.jsx and OrderDetail.jsx

const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    children: [
      { index: true, element: <Dashboard /> },
      {
        path: '/orders',
        lazy: () => import('./routes/Orders.jsx')
      },
      {
        path: '/orders/:id',
        lazy: () => import('./routes/OrderDetail.jsx')
      },
    ],
  },
]);

function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

In this example, both the `/orders` and `/orders/:id` routes are set up to load their respective components lazily.

For the above code to work, there's one important adjustment you must apply to your route component files when using this built-in lazy-loading support: you must replace the default component function export (`export default SomeComponent`) with a named export where the component function is named `Component`.

For example, the `Orders` component code needs to be changed to look like this:

```
import OrdersList from '../components/orders/OrdersList.jsx';
import OrdersSummary from '../components/orders/OrdersSummary.jsx';

function Orders() {
  return (
    <>
      <OrdersSummary />
      <OrdersList />
    </>
  );
}

export const Component = Orders; // named export as "Component"
```

In this code snippet, the `Orders` component function is exported as `Component`. This name is required since React Router looks for a component function named `Component` when activating a lazy-loaded route.

Note

The code for this example can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/examples/12-lazy-loading`.

As explained in *Chapter 10, Behind the Scenes of React and Optimization Opportunities*, adding lazy loading can improve your React application's performance considerably. You should always consider using lazy loading, but you should not use it for every route. It would be especially illogical for routes that are guaranteed to be loaded early, for instance. In the previous example, it would not make too much sense to lazy load the `Dashboard` component since that's the default route (with a path of `/`).

But routes that are not guaranteed to be visited at all (or at least not immediately after the website is loaded) are great candidates for lazy loading.

Summary and Key Takeaways

- Routing is a key feature for many React apps.
- With routing, users can visit multiple pages despite being on an **SPA**.
- The most common package that helps with routing is the React Router library (`react-router-dom`).
- Routes are defined with the help of the `createBrowserRouter()` function and the `RouterProvider` component (typically in the `App` component or the `main.jsx` file, but you can do it anywhere).
- Route definition objects are typically set up with a `path` (for which the route should become active) and an `element` (the content that should be displayed) property.
- Content and markup can be shared across multiple routes by setting up layout routes—i.e., routes wrapping other nested routes.
- Users can navigate between routes by manually changing the URL path, by clicking links, or because of programmatic navigation.
- Internal links (i.e., links leading to application routes defined by you) should be created via the `Link` or `NavLink` components, while links to external resources use the standard `<a>` element.
- Programmatic navigation is triggered via the `navigate()` function, which is yielded by the `useNavigate()` Hook.

- You can define static and dynamic routes: static routes are the default, while dynamic routes are routes where the path (in the route definition) contains a dynamic segment (denoted by a colon, e.g., `:id`).
- The actual values for dynamic path segments can be extracted via the `useParams()` Hook.
- You can use lazy loading to load route-specific code only when the route is actually visited by the user.

What's Next?

Routing is a feature that's not supported by React out of the box but still matters for most React applications. That's why it's included in this book and why the React Router library exists. Routing is a crucial concept that completes your knowledge about the most essential React ideas and concepts, allowing you to build both simple and complex React applications.

The next chapter builds upon this chapter and dives even deeper into React Router, exploring its data fetching and manipulation capabilities.

Test Your Knowledge!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to the examples that can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/exercises/questions-answers.md`:

1. How is routing different from loading content conditionally?
2. How are routes defined?
3. How should you add links to different routes to your pages?
4. How can dynamic routes (e.g., details for one of many products) be added to your app?
5. How can dynamic route parameter values be extracted (e.g., to load product data)?
6. What's the purpose of nested routes?

Apply What You Learned

Apply your knowledge about routing to the following activities.

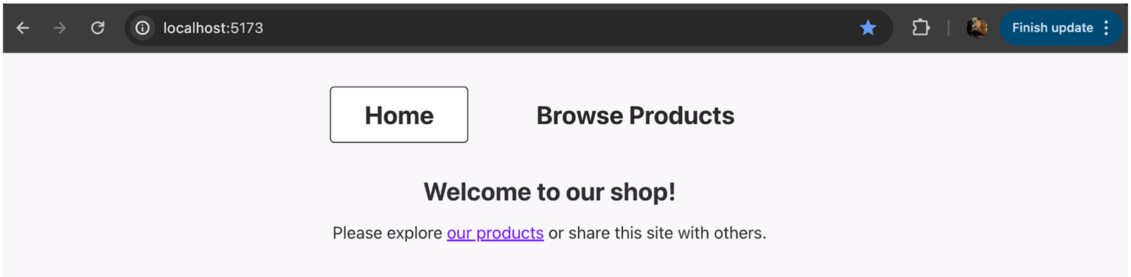
Activity 13.1: Creating a Basic Three-Page Website

In this activity, your task is to create a very basic first draft for a brand-new online shop website. The website must support three main pages:

- A welcome page
- A products overview page that shows a list of available products
- A product details page, which allows users to explore product details

Final website styling, content, and data will be added by other teams, but you should provide some dummy data and default styling. You must also add a shared main navigation bar at the top and implement route-based lazy loading.

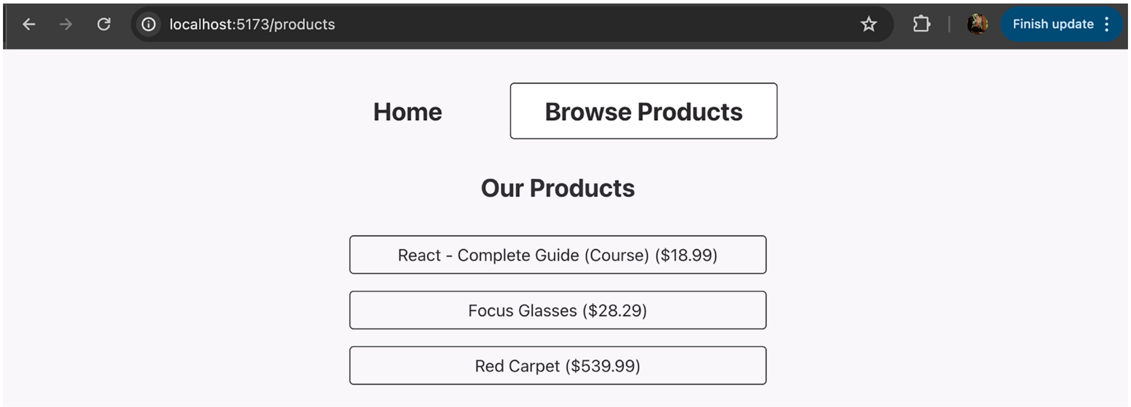
The finished pages should look like this:


localhost:5173
Finish update
Home
Browse Products

Welcome to our shop!

Please explore our products or share this site with others.

Figure 13.12: The welcome page.


localhost:5173/products
Finish update
Home
Browse Products

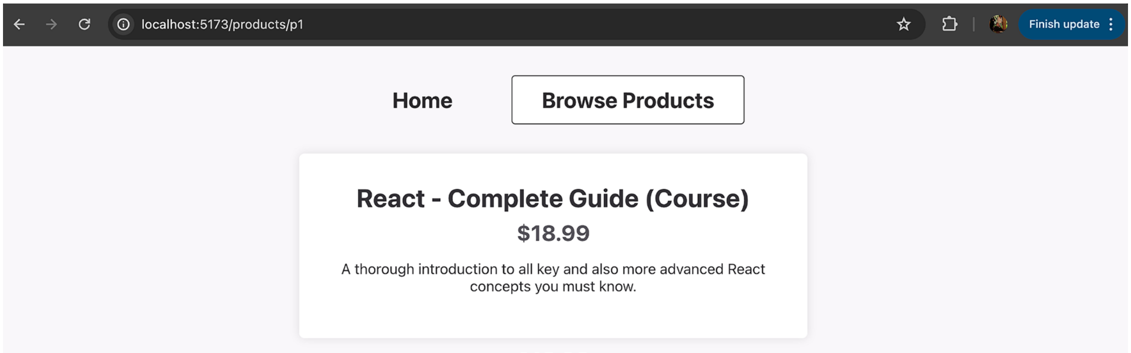
Our Products

React - Complete Guide (Course) ($18.99)

Focus Glasses ($28.29)

Red Carpet ($539.99)

Figure 13.13: A page showing some dummy product placeholders


localhost:5173/products/p1
Finish update
Home
Browse Products

React - Complete Guide (Course)

$18.99

A thorough introduction to all key and also more advanced React concepts you must know.

Figure 13.14: The final product details page with some placeholder data and styles

Note

For this activity, you can, of course, write all CSS styles on your own. But if you want to focus on the React and JavaScript logic, you can also use the finished CSS file from the solution at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/13-routing/activities/practice-1/src/index.css`.

If you use that file, explore it carefully to ensure you understand which IDs or CSS classes might need to be added to certain JSX elements of your solution. You can also use the solution's dummy data instead of creating your own dummy product data. You will find the data for this at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/13-routing/activities/practice-1/src/data/products.js`.

To complete the activity, the solution steps are as follows:

1. Create a new React project and install the React Router package.
2. Create components (with the content shown in the preceding screenshot) that will be loaded for the three required pages.
3. Enable routing and add the route definitions for the three pages.
4. Add a main navigation bar that's visible for all pages.
5. Add all required links and ensure that the navigation bar links reflect whether or not a page is active.
6. Implement lazy loading (for routes where it makes sense).

Note

The full code, and solution, for this activity can be found here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/activities/practice-1`.

14

Managing Data with React Router

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Use React Router to fetch or send data without using `useEffect()` or `useState()`
- Share data between different routes without using React's context feature
- Update the UI based on the current data submission status
- Create page and action routes
- Improve the user experience by deferring the loading of non-critical data

Introduction

In the preceding chapter, you learned how to use React Router to load different components for different URL paths. This is an important feature as it allows you to build multipage websites while still using React.

Routing is a crucial feature for many web applications, and React Router is therefore a very important package. But just as most websites need routing, almost all websites need to fetch and manipulate data. For example, HTTP requests in most websites are sent to load data (such as a list of products or blog posts) or to mutate data (for example, to create a product or a blog post).

In *Chapter 8*, *Handling Side Effects*, you learned that you can use the `useEffect()` Hook and various other React features to send HTTP requests from inside a React application. But if you're using React Router, you get some new, even more powerful tools for working with data.

This chapter will explore which new features are made available by React Router and how they may be used to simplify the process of fetching or sending data.

Data Fetching and Routing Are Tightly Coupled

As mentioned previously, most websites do need to fetch (or send) data, and most websites do need more than one page. But it's important to realize that these two concepts are typically closely related.

Whenever a user visits a new page (such as `/posts`), it's likely that some data will need to be fetched. In the case of a `/posts` page, the required data is probably a list of blog posts that is retrieved from a backend server. The rendered React component (such as `Posts`) must therefore send an HTTP request to the backend server, wait for the response, handle the response (as well as potential errors), and, ultimately, display the fetched data.

Of course, not all pages need to fetch data. Landing pages, "About Us" pages, and "Terms & Use" pages probably don't need to fetch data when a user visits them. Instead, data on those pages is likely to be static. It might even be included in the source code as it doesn't change frequently.

But many pages do need to get data from a backend every time they're loaded—for instance, "Products," "News," "Events," or other infrequently updated pages like the "User Profile."

And data fetching isn't everything. Most websites also contain features that require data submission—be it a blog post that can be created or updated, product data that's administered, or a user comment that can be added. Hence, sending data to a backend is also a very common use case.

And beyond requests, components might also need to interact with other browser APIs, such as `localStorage`. For example, user settings might need to be fetched from storage as a certain page loads.

Naturally, all these interactions happen on pages. But it might not be immediately obvious how tightly data fetching and submission are coupled to routing.

Most of the time, data is fetched when a route becomes active, i.e., when a component (the page component) is rendered for the first time. Sure, users might also be able to click a button to refresh the data, but while this is optional, data fetching upon initial page load is almost always required.

And when it comes to sending data, there is also a close connection to routing. At first sight, it's not clear how it's related because, while it makes sense to fetch data upon page load, it's less likely that you will need to send some data immediately (except perhaps tracking or analytics data).

But it's very likely that *after* sending data, you will want to navigate to a different page, meaning that it's actually the other way around, and instead of initiating data fetching as a page loads, you want to load a different page after sending some data. For example, after an administrator enters some product data and submits the form, they should typically be redirected to a different page (for example, from `/products/new` to the `/products` page).

The connection between data fetching, submission, and routing can therefore be summarized by the following points:

- **Data fetching** often should be initiated when a route becomes active (if that page needs data)
- After **submitting data**, the user should often be redirected to another route

Because these concepts are tightly coupled, React Router provides extra features that vastly simplify the process of working with data.

Sending HTTP Requests without React Router

Working with data is not just about sending HTTP requests. As mentioned in the previous section, you may also need to store or retrieve data via `localStorage` or perform some other operation as a page gets loaded. But sending HTTP requests is an especially common scenario and will therefore be the main use case considered for the majority of this chapter. Nonetheless, it's vital to keep in mind that what you learn in this chapter is not limited to sending HTTP requests.

As you will see, React Router provides various features that help with sending HTTP requests (or using other data fetching and manipulation APIs), but you can also send HTTP requests (or interact with `localStorage` or other APIs) without these features. Indeed, *Chapter 8*, *Handling Side Effects*, already taught you how HTTP requests can be sent from inside React components with the help of `useEffect()`.

When using React Router's data fetching capabilities, you can get rid of `useEffect()` and manual state management.

Note

Besides jumping back in this book, you can also revisit how data fetching with `useEffect()` works via this code example on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/14-routing-data/examples/01-data-fetching-classic`.

Loading Data with React Router

With React Router, fetching data can be simplified down to this, shorter, code snippet:

```
import { useLoaderData } from 'react-router-dom';

function Posts() {
  const loadedPosts = useLoaderData();

  return (
    <main>
      <h1>Your Posts</h1>
      <ul className="posts">
        {loadedPosts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </main>
  );
}

export default Posts;
export async function loader() {
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts'
  );
  if (!response.ok) {
    throw new Error('Could not fetch posts');
  }
  return response;
}
```

Believe it or not, it really *is* that much less code than in the examples shown in *Chapter 8*. Back then, when using `useEffect()`, separate state slices had to be managed to handle loading and error states as well as the received data. Though, to be fair, the content that should be displayed in case of an error is missing here. It's in a separate file (which will be shown later), but it would only add three extra lines of code.

In the preceding code snippet, you see a couple of new features that haven't been covered yet in the book. The `loader()` function and the `useLoaderData()` Hook are added by React Router. These features, along with many others that will be explored throughout this chapter, are made available by the React Router package.

With that library installed, you can set an extra `loader` prop on your route definitions. This prop accepts a function that will be executed by React Router whenever this route (or one of its child routes, if defined) is activated:

```
{ path: '/posts', element: <Posts />, loader: () => {...} }
```

This function can be used to perform any data fetching or other tasks required to successfully display the page component. The logic for getting that required data can therefore be extracted from the component and moved into a separate function.

Since many websites have dozens or even hundreds of routes, adding these loader functions inline in the route definition objects quickly leads to complex and confusing route definitions. For this reason, you will typically add (and export) the `loader()` function in the same file that contains the component that needs the data.

When setting up the route definitions, you can then import the component and its `loader` function and use it like this:

```
import Posts, { loader as postsLoader } from './components/Posts.jsx';

// … other code …

const router = createBrowserRouter([
  { path: '/posts', element: <Posts />, loader: postsLoader }
]);
```

Assigning an alias (`postsLoader`, in this example) to the imported `loader` function is optional but recommended since you most likely have multiple `loader` functions from different components, which would otherwise lead to name clashes.

> **Note**
>
> Technically, you don't need to name your functions `loader`. You could use any name and assign them as values for the `loader` property in the route definition.
>
> But using `loader` as a function name does not just follow the convention; it also has the advantage that React Router's built-in lazy loading support (covered in the previous chapter) lazy-loads the `loader` function when needed. It fails to do that if you pick any other name.

With this `loader` defined, React Router will execute the `loader()` function whenever a route is activated. To be precise, the `loader()` function is called before the component function is executed (that is, before the component is rendered).

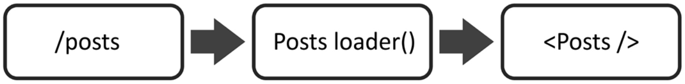


Figure 14.1: The Posts component is rendered after the loader is executed

This also explains why the `Posts` component example at the beginning of this section contained no code that handled any loading state. There simply *is* no loading state since a component function is only executed after its loader has finished (and the data is available). React Router won't finish the page transition until the `loader()` function has finished its job (though, as you will learn toward the end of this chapter, there is a way of changing this behavior).

The `loader()` function can perform any operation of your choice (such as sending an HTTP request, or reaching out to browser storage via the `localStorage` API). Inside that function, you should return the data that should be exposed to the component function. It's also worth noting that the `loader()` function can return any kind of data. It may also return a `Promise` object that then resolves to any kind of data. In that case, React Router will automatically wait for the `Promise` to be fulfilled before executing the related route component function. The `loader()` function can thus perform both asynchronous and synchronous tasks.

> **Note**
>
> It's important to understand that the `loader()` function, like all the other code that makes up your React app, executes on the client side (that is, in the browser of a website visitor). Therefore, you may perform any action that could be performed anywhere else (for example, inside `useEffect()`) in your React app as well.
>
> You must not try to run code that belongs to the server side. Directly reaching out to a database, writing to the file system, or performing any other server-side tasks will fail or introduce security risks, meaning that you might accidentally expose database credentials on the client side.

Getting Access to Loaded Data

Of course, the component that belongs to a `loader` (that is, the component that's part of the same route definition) needs the data returned by the `loader`. This is why React Router offers a new Hook for accessing that data: the `useLoaderData()` Hook.

When called inside a component function, this Hook yields the data returned by the `loader` that belongs to the active route. If that returned data is a `Promise`, React Router (as mentioned earlier) will automatically wait for that `Promise` to resolve and provide the resolved data when `useLoaderData()` is called.

The `loader()` function may also return an HTTP response object (or a `Promise` resolving to a `Response`) object. This is the case in the preceding example because the `fetch()` function yields a `Promise` that resolves to an object of type `Response`. In that instance, React Router automatically extracts the response body and provides direct access to the data that was attached to the response (via `useLoaderData()`).

> **Note**
>
> If a response should be returned, the returned object must adhere to the standard `Response` interface, as defined here: `https://developer.mozilla.org/en-US/docs/Web/API/Response`.
>
> Returning responses might be strange at first. After all, the `loader()` code is still executed inside the browser (not on a server). Therefore, technically, no request was sent, and no response should be required (since the entire code is executed in the same environment, that is, the browser).
>
> For that reason, you can but don't have to return a response; you may return any kind of value. React Router just also supports responses as one possible return value type.

`useLoaderData()` can be called in any component rendered by the currently active route component. That may be the route component itself (`Posts`, in the preceding example), but it may also be any nested component.

For example, `useLoaderData()` can also be used in a `PostsList` component that's included in the `Posts` component (which has a `loader` added to its route definition):

```
import { useLoaderData } from 'react-router-dom';

function PostsList() {
  const loadedPosts = useLoaderData();

  return (
    <main>
      <h1>Your Posts</h1>
      <ul className="posts">
        {loadedPosts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </main>
  );
}
export default PostsList;
```

For this example, the `Posts` component file looks like this:

```
import PostsList from '../components/PostsList.jsx';

function Posts() {
  return (
    <main>
      <h1>Your Posts</h1>
      <PostsList />
    </main>
  );
}

export default Posts;

export async function loader() {
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts'
  );
  if (!response.ok) {
    throw new Error('Could not fetch posts');
  }
  return response;
}
```

This means that `useLoaderData()` can be used in exactly the place where you need the data. The `loader()` function can also be defined wherever you want but it must be added to the route where the data is required.

Note

Depending on the React Router version being used, you might get a warning related to "No HydrateFallback" element being provided. You can ignore this warning as it only matters when using server-side rendering.

Note

You can also explore this code example on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/14-routing-data/examples/02-data-fetching-react-router`.

Loading Data for Dynamic Routes

For most websites, it's unlikely that static, pre-defined routes alone will be sufficient to meet your needs. For instance, if you created a blogging site with exclusively static routes, you would be limited to a simple list of blog posts on `/posts`. To add more details about a selected blog post on routes such as `/posts/1` or `/posts/2` (for posts with different `id` values) you would need to include dynamic routes.

Of course, React Router also supports data fetching with the help of the `loader()` function for dynamic routes:

```
{
  path: "/posts/:id",
  element: <PostDetails />,
  loader: postDetailsLoader
}
```

The `PostDetails` component and its `loader` function can be implemented like this:

```
import { useLoaderData } from 'react-router-dom';

function PostDetails() {
  const post = useLoaderData();
  return (
    <div id="post-details">
      <h1>{post.title}</h1>
      <p>{post.body}</p>
    </div>
  );
}

export default PostDetails;

export async function loader({ params, request }) {
  console.log(request);
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts/' + params.id
  );
```

```
    if (!response.ok) {
      throw new Error('Could not fetch post for id ' + params.id);
    }
    return response;
  }
```

If it looks very similar to the `Posts` component in the *Loading Data with React Router* section, that's no coincidence. Because the `loader()` function works in exactly the same way, there is just one extra feature being used to get hold of the dynamic path segment value: a `params` object that's made available by React Router.

Note

You can also explore this code example on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/14-routing-data/examples/03-dynamic-routes`.

When adding a `loader()` function to a route definition, React Router calls that function whenever the route becomes active, right before the component is rendered. When executing that function, React Router passes an object that contains extra information as an argument to `loader()`.

This object passed to `loader()` includes two main properties:

- A `request` property that contains an object with more details about the request that led to the route activation
- A `params` property that yields an object containing a key-value map of all dynamic route parameters for the active route

The `request` object doesn't matter for this example and will be discussed in the next section. But the `params` object contains an `id` property that carries the `id` value of the post for which the route is loaded. The property is named `id` because, in the route definition, `/posts/:id` was chosen as a path. If a different placeholder name had been chosen, a property with that name would have been available on `params` (for example, for `/posts/:postId`, this would be `params.postId`). This behavior is similar to the `params` object yielded by `useParams()`, as explained in *Chapter 13*, *Multipage Apps with React Router*.

With the help of the `params` object and the post `id`, the appropriate post `id` can be included in the outgoing request URL (for the `fetch()` request), and hence the correct post data can be loaded from the backend API. Once the data arrives, React Router will render the `PostDetails` component and expose the loaded post via the `useLoaderData()` Hook.

Loaders, Requests, and Client-Side Code

In the preceding section, you learned about a `request` object being provided to the `loader()` function. Getting such a `request` object might be confusing because React Router is a client-side library—all the code executes in the browser, not on a server. Therefore, no request should reach the React app (as HTTP requests are sent from the client to the server, not between JavaScript functions on the client side).

And, indeed, there is no request being sent via HTTP. Instead, React Router creates a request object via the browser's built-in `Request` interface to use it as a "data vehicle." This request is not sent via HTTP, but it's used as a value for the `request` property on the data object that is passed to your `loader()` function.

Note

For more information on the built-in `Request` interface, visit `https://developer.mozilla.org/en-US/docs/Web/API/Request`.

This `request` object will be unnecessary in many `loader` functions, but there are occasional scenarios in which you can extract useful information from that object—information that might be needed in the `loader` to fetch the right data.

For example, you can use the `request` object and its `url` property to get access to any search parameters (query parameters) that may be included in the currently active page's URL:

```
export async function loader({ request }) {
  // e.g. for localhost:5173/posts?sort=desc
  const sortDirection = new URL(request.url).searchParams.get('sort');

  // Fetch sorted posts, based on local 'sort' query param value
  const response = await fetch(
    'https://example.com/posts?sorting=' + sortDirection
  );
  return response;
}
```

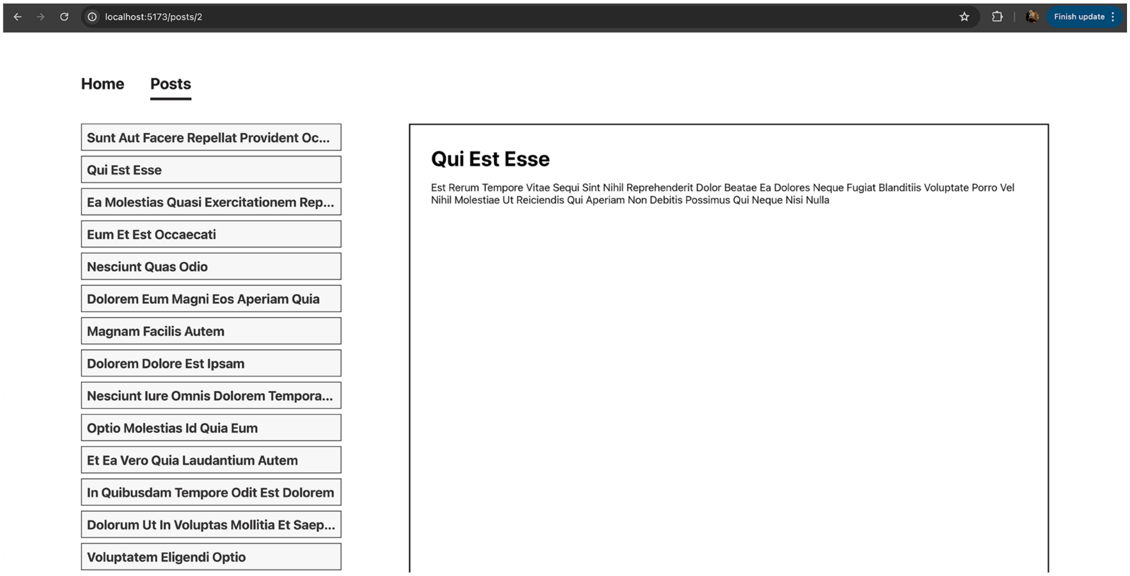
In this code snippet, the `request` value is used to get hold of a query parameter value that's used in the React app URL. That value is then used in an outgoing request.

However, it is vital that you keep in mind that the code inside your `loader()` function, just like all your other React code, always executes on the client side. If, instead, you want to execute code on a server (and, for example, fetch data on the server side), you need to use **server-side rendering** (**SSR**) or some React framework that implements SSR, like Next.js. SSR and Next.js will be covered in the next chapter, *Chapter 15*, *Server-side Rendering & Building Fullstack Apps with Next.js*, and the chapters thereafter.

Layouts Revisited

React Router supports the concept of layout routes. These are routes that contain other routes and render those other routes as nested children. As you may recall, this concept was introduced in *Chapter 13*, *Multipage Apps with React Router*.

Conveniently, layout routes can also be used for sharing data across nested routes. Consider this example website:



- Sunt Aut Facere Repellat Provident Oc...
- Qui Est Esse
- Ea Molestias Quasi Exercitationem Rep...
- Eum Et Est Occaecati
- Nesciunt Quas Odio
- Dolorem Eum Magni Eos Aperiam Quia
- Magnam Facilis Autem
- Dolorem Dolore Est Ipsam
- Nesciunt Iure Omnis Dolorem Tempora...
- Optio Molestias Id Quia Eum
- Et Ea Vero Quia Laudantium Autem
- In Quibusdam Tempore Odit Est Dolorem
- Dolorum Ut In Voluptas Mollitia Et Saep...
- Voluptatem Eligendi Optio

Qui Est Esse

Est Rerum Tempore Vitae Sequi Sint Nihil Reprehenderit Dolor Beatae Ea Dolores Neque Fugiat Blanditiis Voluptate Porro Vel Nihil Molestiae Ut Reiciendis Qui Aperiam Non Debitis Possimus Qui Neque Nisi Nulla

Figure 14.2: A website with a header, a sidebar, and some main content

This website has a header with a navigation bar, a sidebar showing a list of available posts, and a main area that displays the currently selected blog post.

This example includes two layout routes that are nested into each other:

- The root layout route, which includes the top navigation bar that is shared across all pages
- A posts layout route, which includes the sidebar and the main content of its child routes (for example, the details for a selected post)

The route definitions code looks like this:

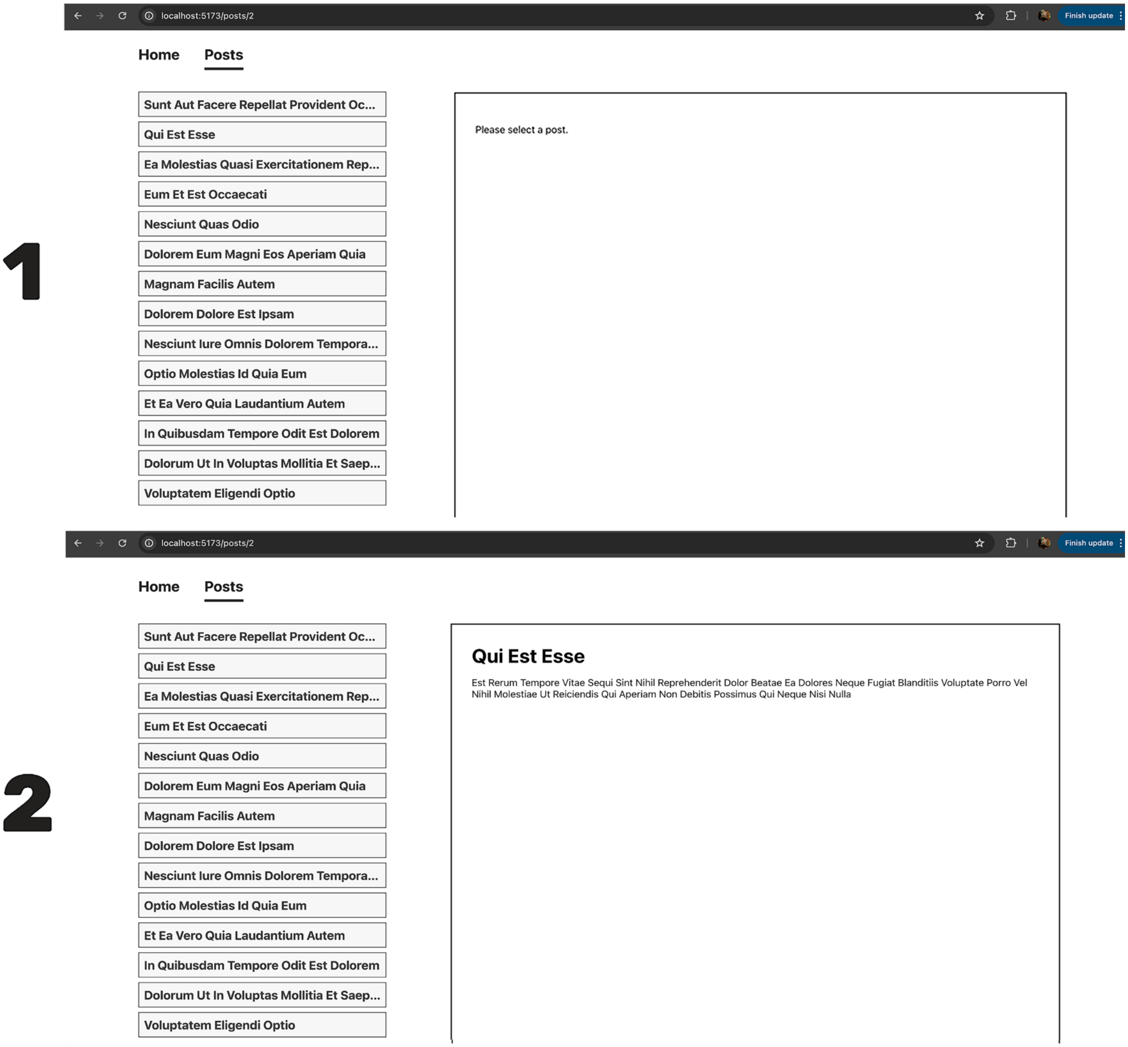
```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />, // main layout, adds navigation bar
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        element: <PostsLayout />, // posts layout, adds posts sidebar
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          {
            path: ':id',
            element: <PostDetails />,
            loader: postDetailsLoader
          },
        ],
      },
    ],
  },
]);
```

With this setup, both the `<Posts />` and the `<PostDetails />` components are rendered next to the sidebar (since the sidebar is part of the `<PostsLayout />` element).

The interesting part is that the `/posts` route (i.e., the layout route) loads the post data, as it has the `postsLoader` assigned to it, and so the `PostsLayout` component file looks like this:

```
import { Outlet, useLoaderData } from 'react-router-dom';

import PostsList from '../components/PostsList.jsx';

function PostsLayout() {
  const loadedPosts = useLoaderData();
  return (
    <div id="posts-layout">
      <nav>
        <PostsList posts={loadedPosts} />
      </nav>
      <main>
        <Outlet />
      </main>
    </div>
  );
}

export default PostsLayout;

export async function loader() {
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts'
  );
  if (!response.ok) {
    throw new Error('Could not fetch posts');
  }
  return response;
}
```

Since layout routes are also regular routes, you can add `loader()` functions and use `useLoaderData()` just as you could in any other route. However, because layout routes are activated for multiple child routes, their data is also displayed for different routes. In the preceding example, the list of blog posts is always displayed on the left side of the screen, no matter if a user visits `/posts` or `/posts/10`:



Sunt Aut Facere Repellat Provident Oc...
Qui Est Esse
Ea Molestias Quasi Exercitationem Rep...
Eum Et Est Occaecati
Nesciunt Quas Odio
Dolorem Eum Magni Eos Aperiam Quia
Magnam Facilis Autem
Dolorem Dolore Est Ipsam
Nesciunt Iure Omnis Dolorem Tempora...
Optio Molestias Id Quia Eum
Et Ea Vero Quia Laudantium Autem
In Quibusdam Tempore Odit Est Dolorem
Dolorum Ut In Voluptas Mollitia Et Saep...
Voluptatem Eligendi Optio

Please select a post.

Sunt Aut Facere Repellat Provident Oc...
Qui Est Esse
Ea Molestias Quasi Exercitationem Rep...
Eum Et Est Occaecati
Nesciunt Quas Odio
Dolorem Eum Magni Eos Aperiam Quia
Magnam Facilis Autem
Dolorem Dolore Est Ipsam
Nesciunt Iure Omnis Dolorem Tempora...
Optio Molestias Id Quia Eum
Et Ea Vero Quia Laudantium Autem
In Quibusdam Tempore Odit Est Dolorem
Dolorum Ut In Voluptas Mollitia Et Saep...
Voluptatem Eligendi Optio

Qui Est Esse

Est Rerum Tempore Vitae Sequi Sint Nihil Reprehenderit Dolor Beatae Ea Dolores Neque Fugiat Blanditiis Voluptate Porro Vel Nihil Molestiae Ut Reiciendis Qui Aperiam Non Debitis Possimus Qui Neque Nisi Nulla

Figure 14.3: The same layout and data are used for different child routes

In this screenshot, the layout and data used do not change as different child routes are activated. React Router also avoids unnecessary data re-fetching (for the blog posts list data) as you switch between child routes. It's smart enough to realize that the surrounding layout hasn't changed.

Reusing Data across Routes

Layout routes do not just help you share components and markup. They also allow you to load and share data across a layout route and its child routes.

For example, the `PostDetails` component (that is, the component that's rendered for the `/posts/:id` route) needs the data for a single post, and that data can be retrieved via a `loader` attached to the `/posts/:id` route:

```
export async function loader({ params }) {
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts/' + params.id
  );
  if (!response.ok) {
    throw new Error('Could not fetch post for id ' + params.id);
  }
  return response;
}
```

This example was discussed earlier in this chapter in the *Loading Data for Dynamic Routes* section. This approach is fine, but in some situations, this extra HTTP request can be avoided. For example, the following route configuration can be simplified, and the extra `postDetailsLoader` on the child route can be avoided:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />, // main layout, adds navigation bar
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        element: <PostsLayout />, // posts layout, adds posts sidebar
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          {
            path: ':id',
            element: <PostDetails />,
            loader: postDetailsLoader // can be removed
          },
        ],
      },
    ],
  },
]);
```

In this example, the `PostsLayout` route already fetches a list of all posts. That layout component is also active for the `PostDetails` route. In such a scenario, fetching a single post is unnecessary, since all the data has already been fetched for the list of posts. Of course, a specific `postDetailsLoader` loader for the `PostDetails` child route would be required if the request for the list of posts (by `postsLoader` on the `PostsLayout` route) didn't yield all the data required by `PostDetails`.

But if all the data is available, React Router allows you to tap into the loader data of a parent route component via the `useRouteLoaderData()` Hook.

This Hook can be used like this:

```
const posts = useRouteLoaderData('posts');
```

`useRouteLoaderData()` requires a route identifier as an argument. It requires an identifier assigned to the ancestor route that contains the data that should be reused. You can assign such an identifier via the `id` property to your routes as part of the route definitions code:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />, // main layout, adds navigation bar
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        id: 'posts', // the id value is up to you
        element: <PostsLayout />, // posts layout, adds posts sidebar
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          {
            path: ':id',
            element: <PostDetails />, // details loader was removed
          },
        ],
      },
    ],
  },
]);
```

The `useRouteLoaderData()` Hook then returns the same data `useLoaderData()` yields in that route to which you added the `id`. In this example, it would provide a list of blog posts.

In `PostDetails`, this Hook can therefore be used like this:

```
import { useParams, useRouteLoaderData } from 'react-router-dom';

function PostDetails() {
  const params = useParams();
  const posts = useRouteLoaderData('posts');
  const post = posts.find((post) => post.id.toString() === params.id);
  return (
    <div id="post-details">
      <h1>{post.title}</h1>
      <p>{post.body}</p>
    </div>
  );
}

export default PostDetails;
```

The `useParams()` Hook is used to get access to the dynamic route parameter value, and the `find()` method is used on the list of posts to identify a single post with a fitting `id` property. In this example, you would thus avoid sending an unnecessary HTTP request by reusing data that's already available.

Therefore, the `postDetailsLoader` that was part of the `/posts/:id` route definition can be removed.

Handling Errors

In the first example at the very beginning of this chapter (where the HTTP request was sent with the help of `useEffect()`), the code did not just handle the success case but also possible errors. In all the React Router-based examples since then, error handling has been omitted. Error handling was not discussed up to this point because, while React Router plays an important role in error handling, it's vital to first gain a solid understanding of how React Router works in general and how it helps with data fetching. But, of course, errors can't always be avoided and definitely should not be ignored.

Thankfully, handling errors is also very straightforward and easy when using React Router's data capabilities. You can set an `errorElement` property on your route definitions and define the element that should be rendered when an error occurs:

```
// ... other imports
import Error from './components/Error.jsx';

const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    errorElement: <Error />,
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        id: 'posts',
        element: <PostsLayout />,
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          { path: ':id', element: <PostDetails /> },
        ],
      },
    ],
  },
]);
```

This `errorElement` property can be set on any route definition of your choice, or even multiple route definitions simultaneously. React Router will render the `errorElement` of the route closest to the place where the error was thrown.

In the preceding snippet, no matter which route produced an error, it would always be the root route's `errorElement` that was displayed (since that's the only route definition with an `errorElement`). But if you also added an `errorElement` to the `/posts` route, and the `:id` route produced an error, it would be the `errorElement` of the `/posts` route that was shown on the screen, as follows:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    errorElement: <Error />, // for all errors not handled elsewhere
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        id: 'posts',
        element: <PostsLayout />,
        // used if /posts or /posts/:id throws an error
        errorElement: <PostsError />, // handles /posts related errors
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          { path: ':id', element: <PostDetails /> },
        ],
      },
    ],
  },
]);
```

This allows you, the developer, to set up fine-grained error handling.

Inside the component used as a value for the `errorElement`, you can get access to the error that was thrown via the `useRouteError()` Hook:

```
import { useRouteError } from 'react-router-dom';

function Error() {
  const error = useRouteError();

  return (
    <>
      <h1>Oh no!</h1>
      <p>An error occurred</p>
      <p>{error.message}</p>
    </>
  );
}

export default Error;
```

With this simple yet effective error-handling solution, React Router allows you to avoid managing error states yourself. Instead, you simply define a standard React element (via the `element` prop) that should be displayed when things go right and an `errorElement` to be displayed if things go wrong.

Onward to Data Submission

Thus far, you've learned a lot about data fetching. But as mentioned earlier in this chapter, React Router also helps with data submission.

Consider the following example component:

```
function NewPost() {
  return (
    <form id="post-form">
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" name="title" />
      </p>
      <p>
        <label htmlFor="text">Text</label>
        <textarea id="text" name="text" rows={3} />
      </p>
      <button>Save Post</button>
    </form>
  );
}

export default NewPost;
```

This component renders a `<form>` element that allows users to enter the details for a new post. Due to the following route configuration, the component is displayed whenever the `/posts/new` route becomes active:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Root />,
    errorElement: <Error />,
    children: [
      { index: true, element: <Welcome /> },
      {
        path: '/posts',
        id: 'posts',
        element: <PostsLayout />,
        loader: postsLoader,
        children: [
          { index: true, element: <Posts /> },
          { path: ':id', element: <PostDetails /> },
          { path: 'new', element: <NewPost /> },
        ],
      },
    ],
  },
]);
```

Without React Router's data-related features, you might handle form submission like this:

```
function NewPost() {
  const navigate = useNavigate();
  async function submitAction(formData) {
    const enteredTitle = formData.get('title');
    const enteredText = formData.get('text');
    const postData = {
      title: enteredTitle,
      text: enteredText
    };

    await fetch('https://jsonplaceholder.typicode.com/posts', {
      method: 'POST',
      body: JSON.stringify(postData),
      headers: {'Content-Type': 'application/json'}
    });
    navigate('/posts');
  }

  return (
    <form action={submitAction}>
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" name="title" />
      </p>
      <p>
        <label htmlFor="text">Text</label>
        <textarea id="text" rows={3} name="text" />
      </p>
      <button>Save Post</button>

    </form>
  );
}
```

Just as before when fetching data, this requires quite a bit of code and logic to be added to the component function. You must manually extract the submitted data, send the HTTP request, and navigate to a different page after receiving an HTTP response.

In addition, you might also need to manage loading state and potential errors (excluded in the preceding example).

Again, React Router offers some help. Where a `loader()` function can be added to handle data loading, an `action()` function can be defined to handle data submission.

When using the new `action()` function, the preceding example component looks like this:

```
import { Form, redirect } from 'react-router-dom';

function NewPost() {
  return (
    <Form method="post" id="post-form">
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" name="title"/>
      </p>
      <p>
        <label htmlFor="text">Text</label>
        <textarea id="text" rows={3} name="text" />
      </p>
      <button>Save Post</button>
    </Form>
  );
}

export default NewPost;

export async function action({ request }) {
  const formData = await request.formData();
  const enteredTitle = formData.get('title');
  const enteredText = formData.get('text');
  const postData = {
    title: enteredTitle,
    text: enteredText
  };
  await fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify(postData),
    headers: { 'Content-Type': 'application/json' },
  });
  return redirect('/posts');
}
```

This code might be similar in length but it has the advantage of moving all the data submission logic out of the component function into a special `action()` function.

Besides the addition of the `action()` function, the example code snippet includes the following important changes and features:

- A `<Form>` component that's used instead of `<form>`.
- The `method` prop is set on the `<Form>` (to `"post"`).
- The submitted data is extracted as `FormData` by calling `request.formData()`.
- The user is redirected via a newly added `redirect()` function (instead of `useNavigate()` and `navigate()`).

But what are these elements about?

Working with action() and Form Data

Just like `loader()`, `action()` is a special function that can be added to route definitions, as follows:

```
import NewPost, { action as newPostAction } from './components/NewPost.jsx';

// ...

{ path: 'new', element: <NewPost />, action: newPostAction },
```

With the `action` prop set on a route definition, the assigned function is automatically called whenever a `<Form>` (not `<form>`!) targeting this route is submitted. `Form` is a component provided by React Router that should be used instead of the default `<form>` element.

Internally, `Form` uses the default `<form>` element but prevents the browser default of creating and sending an HTTP request upon form submission. Instead, React Router creates a `FormData` object and calls the `action()` function defined for the route that's targeted by the `<Form>`, passing a request object, based on the built-in `Request` interface, to it. The passed request object contains the form data generated by React Router. Later in this chapter, in the *Controlling Which <Form> Triggers Which Action* section, you'll learn how to control which `action()` function of which route will be executed by React Router.

Note

Handling form submissions with the help of "actions" might sound familiar—*Chapter 9, Handling User Input & Forms with Form Actions*, discussed a similar concept.

But whereas *Chapter 9* discussed a feature built into React (which was not related or dependent on routing), this chapter explores a core concept of React Router.

Ultimately, you can use either approach for handling form submissions. Or you could use none of the two and instead handle the `submit` event manually via `onSubmit`.

But when using routing with React Router, you'll often end up with cleaner, more concise code that integrates smoothly with other routing features like redirects when using React Router's `<Form>` component and `action()` function.

The form data object that is created by calling `request.formData()` includes all form input values entered into the submitted form. To be registered, an input element such as `<input>`, `<select>`, or `<textarea>` must have the `name` attribute assigned to it. The values set for those `name` attributes can later be used to extract the entered data.

The `request` object (that contains the form data) received by the `action()` function is created by React Router when the form is submitted.

The `Form` component defines the HTTP method of the request object. By setting the `Form`'s `method` prop to either `"get"` (the default) or `"post"`, you control what happens when the form is submitted. When setting `method="get"` (or when not setting `method` at all), a regular URL navigation will occur—just as if a link to a certain path were clicked. Any entered form values will be encoded as URL search parameters in that case. To trigger an `action()` function, `<Form>`'s `method` must be set to `"post"` instead.

However, it's important to understand that the request is not sent via HTTP since `action()`, just like `loader()` or the component function, still executes in the browser rather than on a server.

The `action()` function then receives an object with a `request` property that contains the created request object with the included form data. This `request` object can be used to extract the values entered into the form input fields like this:

```
export async function action({ request }) {
  const formData = await request.formData();
  const postData = Object.fromEntries(formData);

  // ...
}
```

The built-in `formData()` method yields a `Promise` that resolves to a `FormData` object that offers a `get()` method that can be used to get an entered value by its identifier (that is, by the `name` attribute value set on the input element). For example, the value entered into `<input name="title">` could be retrieved via `formData.get('title')`.

Alternatively, you can follow the approach chosen in the preceding code snippet and convert the `formData` object to a simple key-value object via `Object.fromEntries(formData)`. This object (`postData`, in the preceding example) contains the names set on the form input elements as properties and the entered values as values for those properties (meaning that `postData.title` would yield the value entered in `<input name="title">`).

Note

React Router also supports the other main HTTP verbs (`"patch"`, `"put"`, and `"delete"`), and setting `method` to one of these verbs will indeed also trigger the `action()` function.

This can be useful when working with multiple forms that should trigger the same `action()`. By using different methods, you can use one single action to run different code based on the value extracted from `request.method` inside the `action()` function.

But it's worth noting that using methods other than `'get'` and `'post'` is not in line with the HTML standard. Therefore, React Router could remove support for these methods in the future.

Hence, when working with multiple forms that trigger the same `action()`, a more stable solution can be to include a hidden input field with a unique identifier (e.g., `<input type="hidden" name="_method" value="DELETE">`). This value can then be extracted and used (e.g., in an `if` statement) in the `action()` function.

The extracted data can then be used for any operations of your choice. That could be an extra validation step or an HTTP request sent to some backend API, where the data may get stored in a database or file:

```
export async function action({ request }) {
  const formData = await request.formData();
  const postData = Object.fromEntries(formData);
  await fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify(postData),
    headers: { 'Content-Type': 'application/json' },
  });
  return redirect('/posts');
}
```

Finally, once all intended steps are performed, the `action()` function must return a value—any value of any type, but at least `null`. Not returning anything (i.e., omitting the `return` statement) is not allowed. Though, as with the `loader()` function, you may also return a response, for example, a redirect response like this:

```
export async function action({ request }) {
  // action logic …
  return new Response("", {
    status: 302,
    headers: {
      Location: '/posts'
    }
  });
}
```

Indeed, for actions, it's highly likely that you will want to navigate to a different page once the action has been performed (e.g., once an HTTP request to an API has been sent). This may be required to navigate the user away from the data input page to a page that displays all available data entries (for example, from `/posts/new` to `/posts`).

To simplify this common pattern, React Router provides a `redirect()` function that yields a response object that causes React Router to switch to a different route. You can therefore return the result of calling `redirect()` in your `action()` function to ensure that the user is navigated to a different page. It's the equivalent of calling `navigate()` (via `useNavigate()`) when manually handling form submissions.

```
export async function action({ request }) {
  // action logic …
  return redirect('/posts')
}
```

In this snippet, React Router's `redirect()` function is used instead of manually constructing a `Response` object.

Returning Data Instead of Redirecting

As mentioned, your `action()` functions may return anything. You don't have to return a response object. While it is quite common to return a redirect response, you may occasionally want to return some raw data instead.

One scenario in which you might *not* want to redirect the user is after validating the user's input. Inside the `action()` function, before sending the entered data to some API, you may wish to validate the provided values first. If an invalid value (such as an empty title) is detected, a great user experience is typically achieved by keeping the user on the route with the `<Form>`. The values entered by the user shouldn't be cleared and lost; instead, the form should be updated to present useful validation error information to the user. This information can be passed from the `action()` to the component function so that it can be displayed there (for example, next to the form input fields).

In situations like this, you can return a "normal" value (that is, not a redirect response) from your `action()` function:

```
export async function action({ request }) {
  const formData = await request.formData();
  const postData = Object.fromEntries(formData);

  let validationErrors = [];

  if (postData.title.trim().length === 0) {
    validationErrors.push('Invalid post title provided.')
  }
  if (postData.text.trim().length === 0) {
    validationErrors.push('Invalid post text provided.')
  }

  if (validationErrors.length > 0) {
    return validationErrors;
  }

  await fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify(postData),
    headers: { 'Content-Type': 'application/json' },
  });
  return redirect('/posts');
}
```

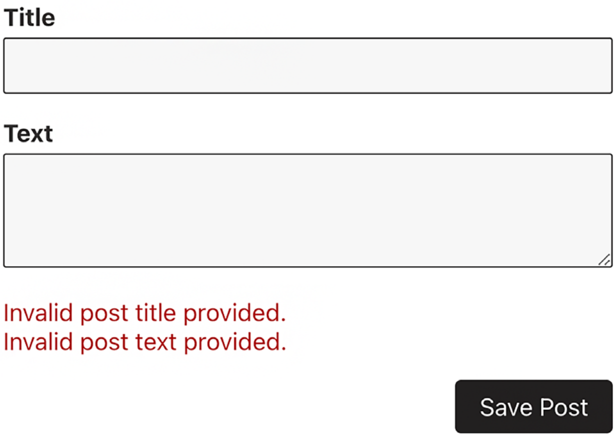
In this example, a `validationErrors` array is returned if the entered `title` or `text` values are empty.

Data returned by an `action()` function can be used in the route component (or any other nested component) via the `useActionData()` Hook:

```
import { Form, redirect, useActionData } from 'react-router-dom';

function NewPost() {
  const validationErrors = useActionData();

  return (
    <Form method="post" id="post-form">
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" name="title" />
      </p>
      <p>
        <label htmlFor="text">Text</label>
        <textarea id="text" name="text" rows={3} />
      </p>
      <ul>
        {validationErrors &&
          validationErrors.map((err) => <li key={err}>{err}</li>)}
      </ul>
      <button>Save Post</button>
    </Form>
  );
}
```

`useActionData()` works a lot like `useLoaderData()`, but unlike `useLoaderData()`, it's not guaranteed to yield any data. This is because while `loader()` functions always get called before the route component is rendered, the `action()` function only gets called once the `<Form>` is submitted.

In this example, `useActionData()` is used to get access to the `validationErrors` returned by `action()`. If `validationErrors` is truthy (that is, is not `undefined`), the array will be mapped to a list of error items that are displayed to the user:



Invalid post title provided.
Invalid post text provided.

Figure 14.4: Validation errors are output below the input fields

The `action()` function is therefore quite versatile in that you can use it to perform an action and redirect away as well as to conduct more than one operation and return different values for different use cases.

Controlling Which <Form> Triggers Which Action

Earlier in this chapter, in the section *Working with action() and Form Data*, you learned that when `<Form>` is used instead of `<form>`, React Router will execute the targeted `action()` function. But which `action()` function is targeted by `<Form>`?

By default, it's the `action()` function assigned to the route that also renders the form (either directly or via some descendent component). Consider this route definition:

```
{ path: '/posts/new', element: <NewPost />, action: newPostAction }
```

With this definition, the `newPostAction()` function would be triggered whenever any `<Form>` inside of the `NewPost` component (or any nested component) is submitted.

In many cases, this default behavior is exactly what you want. But you can also target `action()` functions defined on other routes by setting the `action` prop on `<Form>` to the path of the route that contains the `action()` that should be executed:

```
// form rendered in a component that belongs to /posts
<Form method="post" action="/save-data">
  ...
</Form>
```

This form would cause React Router to execute the `action` belonging to the `/save-data` route—even though the `<Form>` component may be rendered as part of a component that belongs to a different route (e.g., `/posts`).

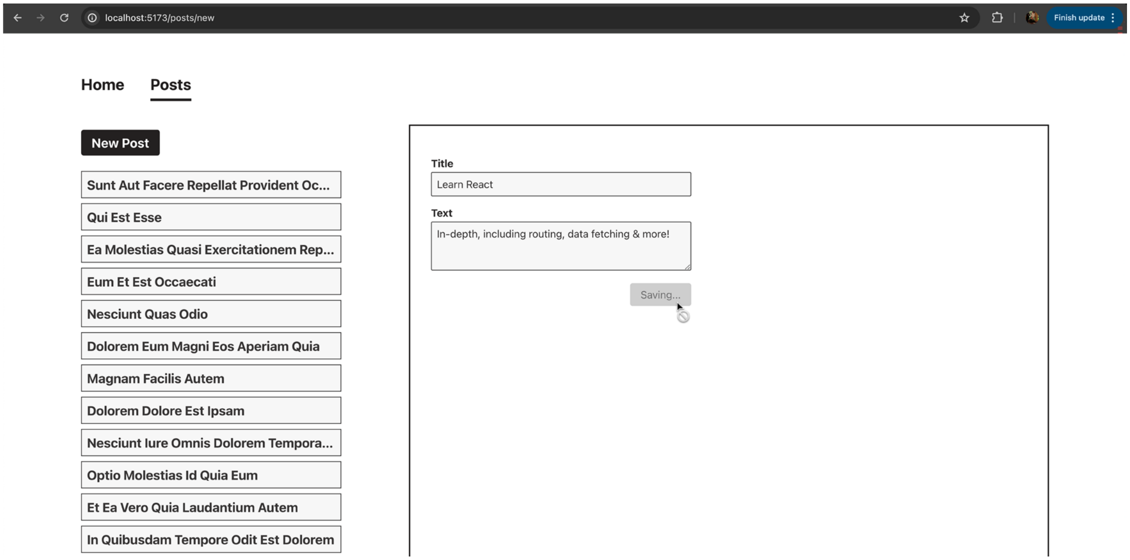
It is worth noting, though, that targeting a different route will lead to a page transition to that route's path, even if your action does not return a redirect response. In a later section of this chapter, entitled *Behind-the-Scenes Data Fetching and Submission*, you will learn how that behavior can be avoided.

Reflecting the Current Navigation Status

After submitting a form, the `action()` function that's triggered may need some time to perform all intended operations. Sending HTTP requests to APIs in particular can take up to a few seconds.

Of course, it's not a great user experience if the user doesn't get any feedback about the current data submission status. It's not immediately clear if anything happened at all after the submit button was clicked.

For that reason, you might want to show a loading spinner or update the button caption while the `action()` function is running. Indeed, one common way of providing user feedback is to disable the submit button and change its caption like this:


localhost:5173/posts/new
Finish update
Home
Posts
New Post
Sunt Aut Facere Repellat Provident Oc...
Qui Est Esse
Ea Molestias Quasi Exercitationem Rep...
Eum Et Est Occaecati
Nesciunt Quas Odio
Dolorem Eum Magni Eos Aperiam Quia
Magnam Facilis Autem
Dolorem Dolore Est Ipsam
Nesciunt Iure Omnis Dolorem Tempora...
Optio Molestias Id Quia Eum
Et Ea Vero Quia Laudantium Autem
In Quibusdam Tempore Odit Est Dolorem
Saving...

Title: Learn React

Text: In-depth, including routing, data fetching & more!

Figure 14.5: The submit button is grayed out

You can get the current React Router status (that is, whether it's currently transitioning to another route or executing an `action()` function) via the `useNavigation()` Hook. This Hook provides a navigation object that contains various pieces of routing-related information.

Most importantly, this object has a `state` property that yields a string describing the current navigation status. This property is set to one of the following three possible values:

- `submitting`: If an `action()` function is currently executing

- `loading`: If a `loader()` function is currently executing (for example, because of a `redirect()` response)
- `idle`: If no `action()` or `loader()` functions are currently being executed

You can therefore use this `state` property to find out whether React Router is currently navigating to a different page or executing an `action()`. Hence, the submit button can be updated as shown in the preceding screenshot via this code:

```
import {
  Form,
  redirect,
  useActionData,
  useNavigation
} from 'react-router-dom';

function NewPost() {
  const validationErrors = useActionData();

  const navigation = useNavigation();
  const isSubmitting = navigation.state !== 'idle';

  return (
    <Form method="post" id="post-form">
      <p>
        <label htmlFor="title">Title</label>
        <input type="text" id="title" name="title" />
      </p>
      <p>
        <label htmlFor="text">Text</label>
        <textarea id="text" name="text" rows={3} />
      </p>
      <ul>
        {validationErrors &&
          validationErrors.map((err) => <li key={err}>{err}</li>)}
      </ul>
      <button disabled={isSubmitting}>
        {isSubmitting ? 'Saving...' : 'Save Post'}
      </button>
    </Form>
  );
}
```

In this example, the `isSubmitting` constant is `true` if the current navigation state is anything but `'idle'`. This constant is then used to disable the submit button (via the `disabled` attribute) and adjust the button's caption.

Submitting Forms Programmatically

In some cases, you won't want to instantly trigger an `action()` when a form is submitted—for example, if you need to ask the user for confirmation first such as when triggering actions that delete or update data.

For such scenarios, React Router allows you to submit a form (and therefore trigger an `action()` function) programmatically. Instead of using the `Form` component provided by React Router, you handle the form submission manually using the default `<form>` element. As part of your code, you can then use a `submit()` function provided by React Router's `useSubmit()` Hook to trigger the `action()` manually once you're ready for it.

Consider this example:

```
import {
  redirect,
  useParams,
  useRouteLoaderData,
  useSubmit,
} from 'react-router-dom';

function PostDetails() {
  const params = useParams();
  const posts = useRouteLoaderData('posts');
  const post = posts.find((post) => post.id.toString() === params.id);

  const submit = useSubmit();

  function handleSubmit(event) {
    event.preventDefault();

    const proceed = window.confirm('Are you sure?');

    if (proceed) {
      submit(
        { message: 'Your submitted data, if needed' },
        {
          method: 'post',
        }
        );
      }
    }

    return (
      <div id="post-details">
        <h1>{post.title}</h1>
        <p>{post.body}</p>
        <form onSubmit={handleSubmit}>
          <button>Delete</button>
        </form>
      </div>
    );
  }

  export default PostDetails;

  // action must be added to route definition!
  export async function action({ request }) {
    const formData = await request.formData();
    console.log(formData.get('message'));
    console.log(request.method);
    return redirect('/posts');
  }
```

In this example, the `action()` is manually triggered by programmatically submitting data via the `submit()` function provided by `useSubmit()`. This approach is required as it would otherwise be impossible to ask the user for confirmation (via the browser's `window.confirm()` method).

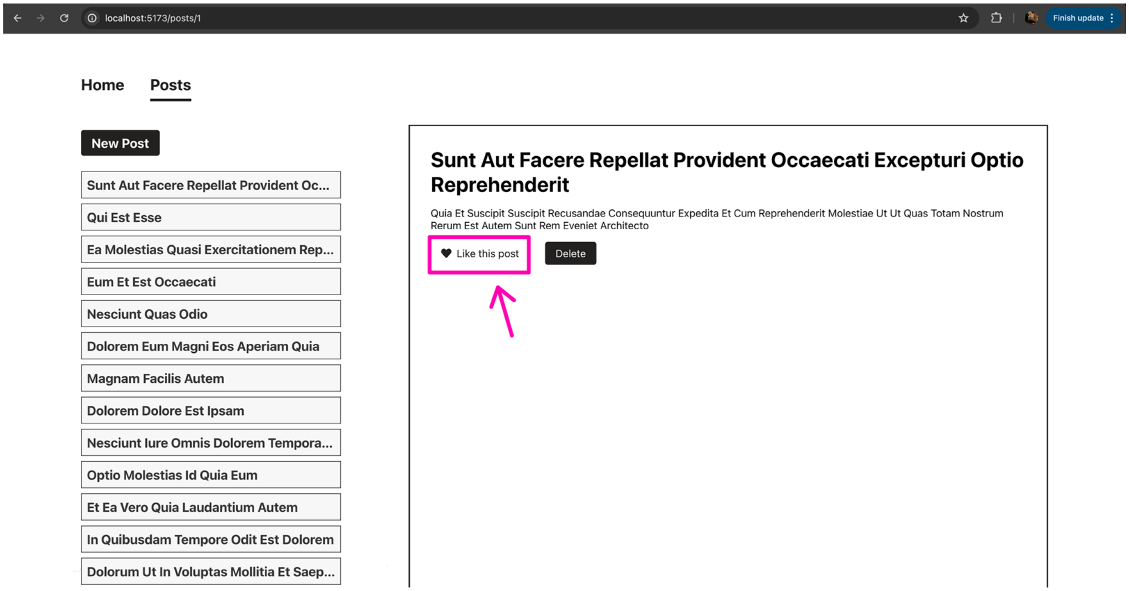
Because data is submitted programmatically, the default `<form>` element should be used and the `submit` event handled manually. As part of this process, the browser's default behavior of sending an HTTP request must also be prevented manually.

Typically, using `<Form>` instead of programmatic submission is preferable. But in certain situations, such as the preceding example, being able to control form submission manually can be useful.

Behind-the-Scenes Data Fetching and Submission

There are also situations in which you may need to trigger an action or load data without causing a page transition.

A Like button would be an example. When it's clicked, a process should be triggered in the background (such as storing information about the user and the liked post), but the user should not be directed to a different page:


localhost:5173/posts/1
Finish update
Home
Posts
New Post
Sunt Aut Facere Repellat Provident Oc...
Qui Est Esse
Ea Molestias Quasi Exercitationem Rep...
Eum Et Est Occaecati
Nesciunt Quas Odio
Dolorem Eum Magni Eos Aperiam Quia
Magnam Facilis Autem
Dolorem Dolore Est Ipsam
Nesciunt Iure Omnis Dolorem Tempora...
Optio Molestias Id Quia Eum
Et Ea Vero Quia Laudantium Autem
In Quibusdam Tempore Odit Est Dolorem
Dolorum Ut In Voluptas Mollitia Et Saep...
Like this post
Delete

Sunt Aut Facere Repellat Provident Occaecati Excepturi Optio Reprehenderit

Quia Et Suscipit Suscipit Recusandae Consequuntur Expedita Et Cum Reprehenderit Molestiae Ut Ut Quas Totam Nostrum Rerum Est Autem Sunt Rem Eveniet Architecto

Figure 14.6: A Like button below a post

To achieve this behavior, you could wrap the button into a `<Form>` and, at the end of the `action()` function, simply redirect back to the page that is already active.

But technically, this would still lead to an extra navigation action. Therefore, `loader()` functions would be executed and other possible side-effects might occur (the current scroll position could be lost, for example). For that reason, you might want to avoid this kind of behavior.

Thankfully, React Router offers a solution: the `useFetcher()` Hook, which yields an object that contains a `submit()` method. Unlike the `submit()` function provided by `useSubmit()`, the `submit()` method yielded by `useFetcher()` is meant for triggering actions (or `loader()` functions) without starting a page transition.

A Like button, as described previously, can be implemented like this (with the help of `useFetcher()`):

```
import {
  // ... other imports
  useFetcher,
} from 'react-router-dom';
import { FaHeart } from 'react-icons/fa';

function PostDetails() {
  // ... other code & logic
```

```
    const fetcher = useFetcher();

    function handleLikePost() {
      fetcher.submit(null, {
        method: 'post',
        action: `/posts/${post.id}/like`,
        // targeting an action on another route
      });
    }

    return (
      <div id="post-details">
        <h1>{post.title}</h1>
        <p>{post.body}</p>
        <div className="actions">
          <button className="icon-btn" onClick={handleLikePost}>
            <FaHeart />
            <span>Like this post</span>
          </button>
          <form onSubmit={handleSubmit}>
            <button>Delete</button>
          </form>
        </div>
      </div>
    );
  }
```

The `fetcher` object returned by `useFetcher()` has various properties. For example, it also contains properties that provide information about the current status of the triggered action or loader (including any data that may have been returned).

But this object also includes two important methods:

- `load()`: To trigger the `loader()` function of a route (e.g., `fetcher.load('/route-path')`)
- `submit()`: To trigger an `action()` function with the provided data and configuration

In the code snippet above, the `submit()` method is called to trigger the action defined on the `/posts/<post-id>/like` route. Without `useFetcher()` (i.e., when using `useSubmit()` or `<Form>`), React Router would switch to the selected route path when triggering its action. With `useFetcher()`, this is avoided, and the action of that route can be called from inside another route (meaning the action defined for `/posts/<post-id>/like` is called while the `/posts/<post-id>` route is active).

This also allows you to define routes that don't render any element (that is, in which there is no page component) and, instead, only contain a `loader()` or `action()` function. For example, the `/posts/<post-id>/like` route file (`pages/like.js`) looks like this:

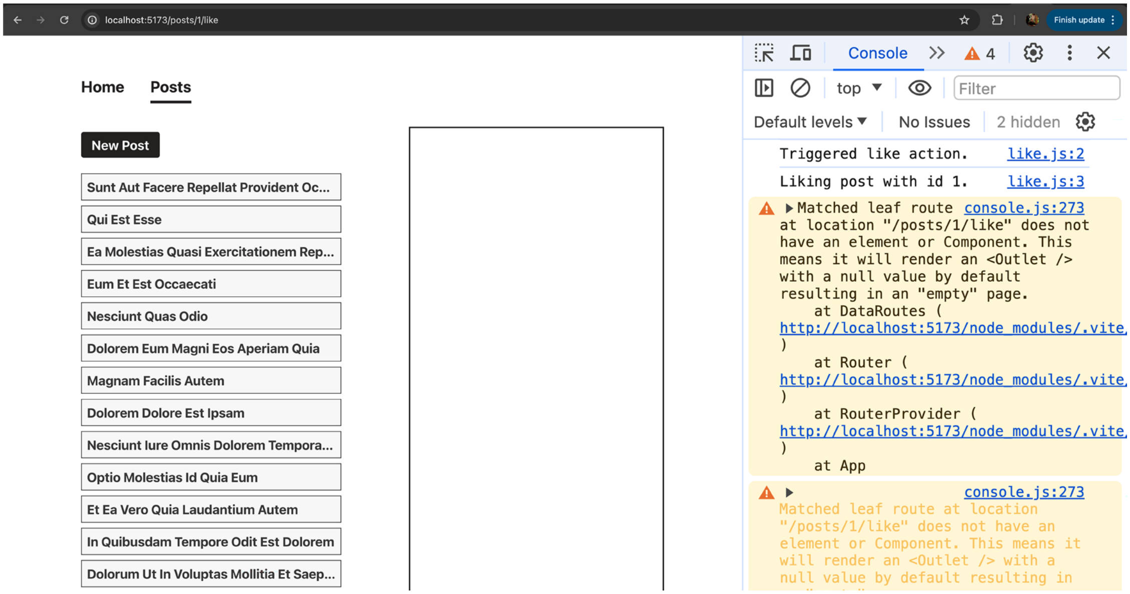
```
// there is no component function in this file!

export function action({ params }) {
  console.log('Triggered like action.');
  console.log(`Liking post with id ${params.id}.`);
  // Do anything else
  // May return data or response, including redirect() if needed
  return null; // something must be returned, even if it's just null
}
```

As mentioned in the code snippet, any data may be returned in this action. But you must at least return `null`—avoiding the `return` statement and not returning anything is not allowed and will cause an error.

It's registered as a route as follows:

```
import { action as likeAction } from './pages/like.js';
// ...
{ path: ':id/like', action: likeAction },
```

This works because this `action()` is only triggered via the `submit()` method provided by `useFetcher()`. `<Form>` and the `submit()` function yielded by `useSubmit()` would instead initiate a route transition to `/posts/<post-id>/like`. Without the `element` property being set on the route definition, this transition would lead to an empty page, as shown here:


localhost:5173/posts/1/like
Finish update
Console
4
top
Filter
Default levels
No Issues
2 hidden
Home
Posts
New Post
Sunt Aut Facere Repellat Provident Oc...
Qui Est Esse
Ea Molestias Quasi Exercitationem Rep...
Eum Et Est Occaecati
Nesciunt Quas Odio
Dolorem Eum Magni Eos Aperiam Quia
Magnam Facilis Autem
Dolorem Dolore Est Ipsam
Nesciunt Iure Omnis Dolorem Tempora...
Optio Molestias Id Quia Eum
Et Ea Vero Quia Laudantium Autem
In Quibusdam Tempore Odit Est Dolorem
Dolorum Ut In Voluptas Mollitia Et Saep...

```
Triggered like action.                    like.js:2
Liking post with id 1.                    like.js:3
▶Matched leaf route                console.js:273
at location "/posts/1/like" does not
have an element or Component. This
means it will render an <Outlet />
with a null value by default
resulting in an "empty" page.
    at DataRoutes (
http://localhost:5173/node_modules/.vite
)
    at Router (
http://localhost:5173/node_modules/.vite
)
    at RouterProvider (
http://localhost:5173/node_modules/.vite
)
    at App
▶                                  console.js:273
Matched leaf route at location
"/posts/1/like" does not have an
element or Component. This means it
will render an <Outlet /> with a
null value by default resulting in
```

Because of the extra flexibility it offers, `useFetcher()` can be very useful when building highly interactive user interfaces. It's not meant as a replacement for `useSubmit()` or `<Form>`, but rather, as an additional tool for situations where no route transition is required or wanted.

Deferring Data Loading

Up to this point in the chapter, all data fetching examples have assumed that a page should only be displayed once all its data has been fetched. That's why there was never any loading state that would have been managed (and hence no loading fallback content that would have been displayed).

In many situations, this is exactly the behavior you want as it does not often make sense to show a loading spinner or similar fallback content for a fraction of a second just to then replace it with the actual page data.

But there are also situations in which the opposite behavior might be desirable—for example, if you know that a certain page will take quite a while to load its data (possibly due to a complex database query that must be executed on the backend) or if you have a page that loads different pieces of data and some pieces are much slower than others.

In such scenarios, it may make sense to render the page component even though some data is still missing. React Router also supports this use case by allowing you to defer data loading, which, in turn, enables the page component to be rendered before the data is available.

Deferring data loading is as simple as returning a promise from the loader (instead of awaiting it there):

```
// ... other imports
export async function loader() {
  return {
    posts: getPosts()
  };
}
```

In this example, `getPosts()` is a function that returns a (slow) `Promise`:

```
async function getPosts() {
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts'
  );
  await wait(3); // utility function, simulating a slow response
  if (!response.ok) {
    throw new Error('Could not fetch posts');
  }
  const data = await response.json();
  return data;
}
```

React Router allows you to return raw promises. When doing so, you can wait for the actual values yielded by those promises in the client-side code.

Inside the component function where `useLoaderData()` is used, you must also use a new component provided by React Router: the `Await` component. It's used like this:

```
import { Suspense } from 'react';
import { Await } from 'react-router-dom';
// ... other imports

function PostsLayout() {
  const data = useLoaderData();

  return (
    <div id="posts-layout">
      <nav>
        <Suspense fallback={<p>Loading posts...</p>}>
          <Await resolve={data.posts}>
            {(loadedPosts) => <PostsList posts={loadedPosts} />}
          </Await>
        </Suspense>
      </nav>
      <main>
        <Outlet />
      </main>
    </div>
  );
}
```

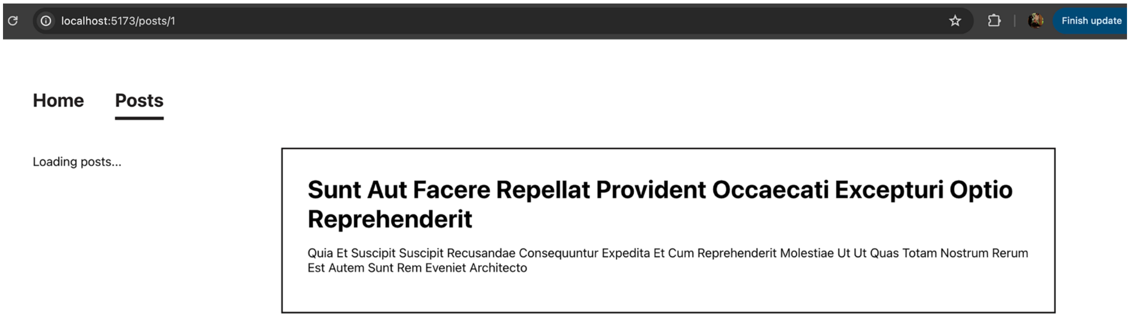
The `<Await>` element takes a `resolve` prop that receives a value of type `Promise` from the loader data. It's wrapped by the `<Suspense>` component provided by React.

The value passed to resolve is a `Promise` that was stored in the object returned by the `loader()` function. There, a key named posts was used to hold that `Promise`. The value for that key was the `Promise` returned by `getPosts()`. It's this `Promise` that's passed as a value to `resolve` via `<Await resolve={data.posts}>`. If a different key name were used (e.g., `blogPosts`), that key name had to be referenced when setting `resolve` (e.g., `<Await resolve={data.blogPosts}>`).

`Await` automatically waits for the `Promise` to resolve before then calling the function that's passed to `<Await>` as a child (that is, the function passed between the `<Await>` opening and closing tags). This function is executed by React Router once the data of the deferred operation is available. Therefore, inside that function, `loadedPosts` is received as a parameter, and the final user interface elements can be rendered.

The `Suspense` component that's used as a wrapper around `<Await>` defines some fallback content that is rendered as long as the deferred data is not yet available. In *Chapter 10, Behind the Scenes of React and Optimization Opportunities*, the `Suspense` component was used to show some fallback content until the missing code was downloaded. Now, it's used to bridge the time until the required data is available.

As shown in *Figure 14.8*, when returning a `Promise` (and using `<Await>`) like this, other parts of the website, that are not loaded via `<Await>`, are already rendered and displayed while waiting for the posts data.



Loading posts...

Sunt Aut Facere Repellat Provident Occaecati Excepturi Optio Reprehenderit

Quia Et Suscipit Suscipit Recusandae Consequuntur Expedita Et Cum Reprehenderit Molestiae Ut Ut Quas Totam Nostrum Rerum Est Autem Sunt Rem Eveniet Architecto

Figure 14.8: Post details are already visible while the list of posts is loading

Another big advantage of returning a `Promise` and awaiting it in the client-side code is that you can easily combine multiple fetching processes and control which processes should be deferred and which ones should not. For example, a route might be fetching different pieces of data. If only one process tends to be slow, you could defer only the slow one like this:

```
export async function loader() {
  return {
      posts: getPosts(), // slow operation => deferred
      userData: await getUserData() // fast operation => NOT deferred
    };
}
```

In this example, `getUserData()` is not deferred because the `await` keyword is added in front of it. Therefore, JavaScript waits for that `Promise` (the `Promise` returned by `getUserData()`) to resolve before returning from `loader()`. Hence, the route component is rendered once `getUserData()` finishes but before `getPosts()` is done.

Summary and Key Takeaways

- React Router can help you with data fetching and submission.
- You can register `loader()` functions for your routes, causing data fetching to be initialized as a route becomes active.
- `loader()` functions return data (or responses, wrapping data) that can be accessed via `useLoaderData()` in your component functions.

- `loader()` data can be used across components via `useRouteLoaderData()`.
- You can also register `action()` functions on your routes that are triggered upon form submissions.
- To trigger `action()` functions, you must use React Router's `<Form>` component or submit data programmatically via `useSubmit()` or `useFetcher()`.
- `useFetcher()` can be used to load or submit data without initiating a route transition.
- When fetching slow data, you can return promises without awaiting them in the `loader()` to defer loading some or all of a route's data.

What's Next?

Fetching and submitting data are extremely common tasks, especially when building more complex React applications.

Typically, those tasks are closely connected to route transitions, and React Router is the perfect tool for handling this kind of operation. That's why the React Router package offers powerful data management capabilities that vastly simplify these processes.

In this chapter, you learned how React Router assists you with fetching or submitting data and which advanced features help you handle both basic and more complex data manipulation scenarios.

Therefore, this chapter concludes the list of core React Router features you need to know.

The next chapters will explore React's server-side capabilities and how you may build fullstack applications with React, load data on a server, and use the Next.js framework.

Test Your Knowledge!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to the examples found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/14-routing-data/exercises/questions-answers.md`:

1. How are data fetching and submission related to routing?
2. What's the purpose of `loader()` functions?
3. What's the purpose of `action()` functions?
4. What's the difference between `<Form>` and `<form>`?
5. What's the difference between `useSubmit()` and `useFetcher()`?
6. What's the idea behind returning promises instead of awaiting them in a `loader()`?

Apply What You Learned

Apply your knowledge about routing, combined with data manipulation, to the following activity.

Activity 14.1: A To-Dos App

In this activity, your task is to create a basic to-do list web app that allows users to manage their daily to-do tasks. The finished page must allow users to add to-do items, update to-do items, delete to-do items, and view a list of to-do items.

The following paths must be supported:

- `/`: The main page, responsible for loading and displaying a list of to-do items
- `/new`: A page, opened as a modal above the main page, allowing users to add a new to-do item
- `/:id`: A page, also opened as a modal above the main page, allowing users to update or delete a selected to-do item

If no to-do items exist yet, a fitting info message should be shown on the `/` page. If users try to visit `/:id` with an invalid to-do ID, an error modal should be displayed.

> **Note**
>
> For this activity, there is no backend API you could use. Instead, use `localStorage` to manage the to-do data. Keep in mind that the `loader()` and `action()` functions are executed on the client side and can therefore use any browser APIs, including `localStorage`.
>
> You will find example implementations for adding, updating, deleting, and getting to-do items from `localStorage` at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/14-routing-data/activities/practice-1/src/data/todos.js`.
>
> Also, don't be confused by the pages that open as modals above other pages. Ultimately, these are simply nested pages, styled as modal overlays. In case you get stuck, you can use the example `Modal` wrapper component found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/14-routing-data/activities/practice-1/src/components/Modal.jsx`.
>
> For this activity, you can write all CSS styles on your own if you so choose. But if you want to focus on the React and JavaScript logic, you can also use the finished CSS file from the solution at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/14-routing-data/activities/practice-1/src/index.css`.
>
> If you use that file, explore it carefully to ensure you understand which IDs or CSS classes might need to be added to certain JSX elements of your solution.

To complete the activity, perform the following steps:

1. Create a new React project and install the React Router package.
2. Create components (with the content shown in the screenshots below) that will be loaded for the three required pages. Also, add links (or programmatic navigation) between these pages.
3. Enable routing and add the route definitions for the three pages.
4. Create `loader()` functions to load (and use) all the data needed by the individual pages.
5. Add `action()` functions for adding, updating, and deleting to-dos.

   Hint: If you need to submit multiple forms for different actions from the same page, you could include a hidden input field that sets some value you can check for in your `action()` function, e.g., `<input type="hidden" name="_method" value="DELETE">`. Alternatively, you can also set `<Form method="delete">` (or set it to `"patch"`, `"put"`, or other HTTP verbs) and check for `request.method` in your `action()` function.

6. Add error handling in case data loading or saving fails.

The finished pages should look like this:

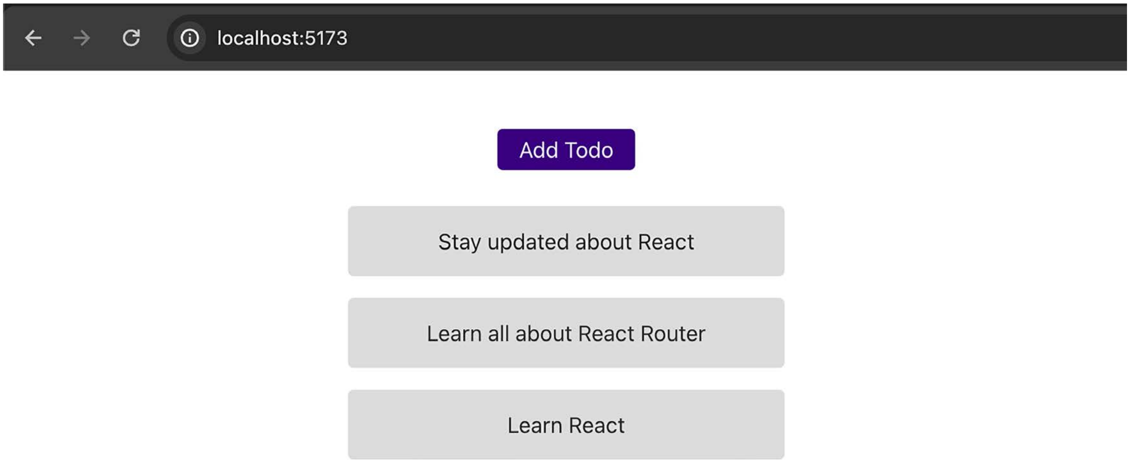

localhost:5173
Add Todo
Stay updated about React
Learn all about React Router
Learn React

Figure 14.9: The main page displaying a list of to-dos

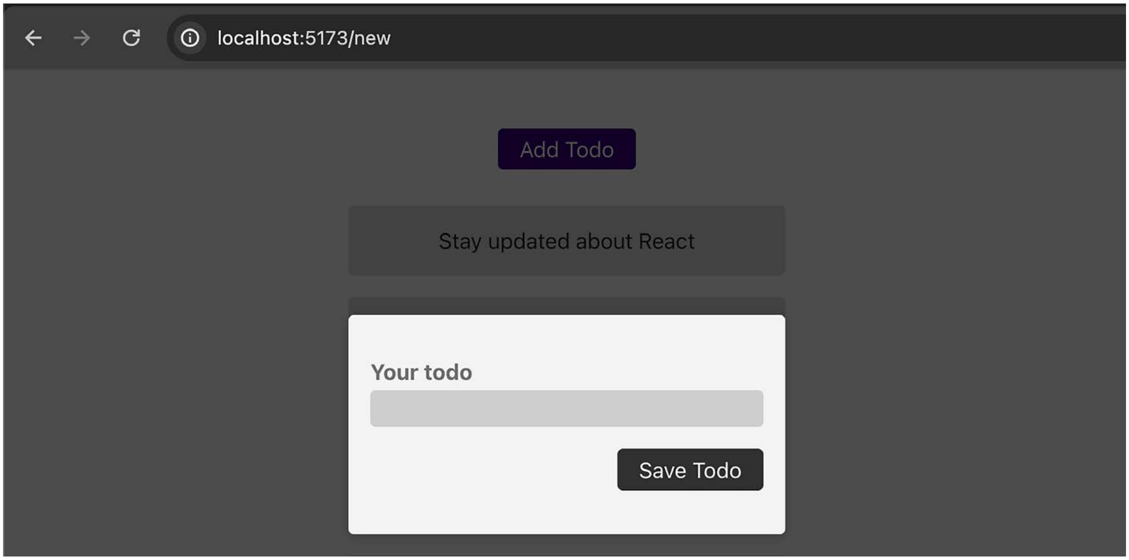
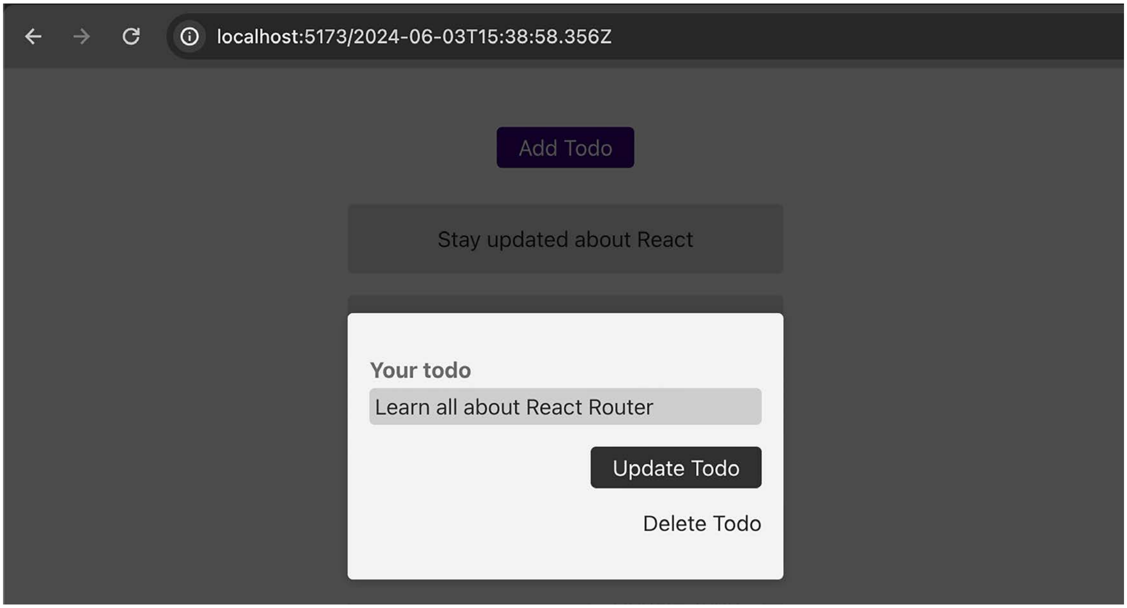

localhost:5173/new
Add Todo
Stay updated about React
Your todo
Save Todo

Figure 14.10: The /new page, opened as a modal, allowing users to add a new to-do

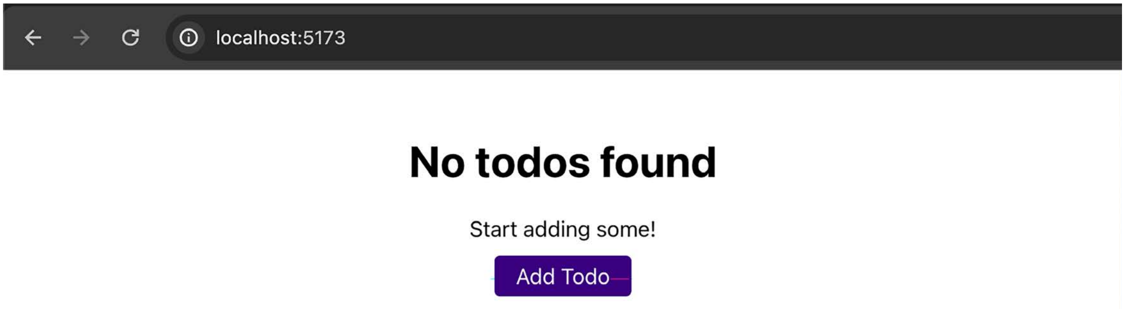

localhost:5173/2024-06-03T15:38:58.356Z
Add Todo
Update Todo
Delete Todo

Stay updated about React

Your todo

Learn all about React Router

Figure 14.11: The /:id page, also opened as a modal, allowing users to edit or delete a to-do


localhost:5173
Add Todo

No todos found

Start adding some!

Figure 14.12: An info message, displayed if no to-dos were found

Note

The full code, and solution, to this activity can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/14-routing-data/activities/practice-1`.

15

Server-side Rendering & Building Fullstack Apps with Next.js

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Describe the difference between client-side and server-side React
- Determine which kind of React app to build
- Build fullstack React apps with the help of the Next.js framework
- Explain the key features and advantages of Next.js

Introduction

Thus far in this book, you have learned a lot about building client-side React applications, that is, applications where the (transpiled) React code executes in the browsers of your website visitors.

This makes sense because React was originally created to simplify the process of building interactive and reactive UIs by running JavaScript code on the client side. To this date, most React features, including the ones covered up to this point in this book (e.g., state, context, and routing), exist to fulfill this purpose.

But, as you will learn in this and the following chapters, you can actually also execute React code on the server side. There are certain React features that may only be used there—for example, React Server Components, which will be covered in great detail in the *Chapter 16, React Server Components & Server Actions*.

This chapter will get you started with React on the server side, briefly explain what **server-side rendering** (**SSR**) is, and introduce you to Next.js, a popular and feature-rich fullstack framework for React that allows you to blend backend and frontend code. You will learn how to create Next.js apps and how to use core Next.js features like file-based routing.

What's the Problem with Client-Side React Apps?

The big advantage of **single-page applications** (**SPAs**) and client-side React is that you can build highly reactive and interactive web UIs. The UI can be updated almost instantly, visible page reloads and switches can be avoided, and hence your users benefit from a mobile-app-like user experience.

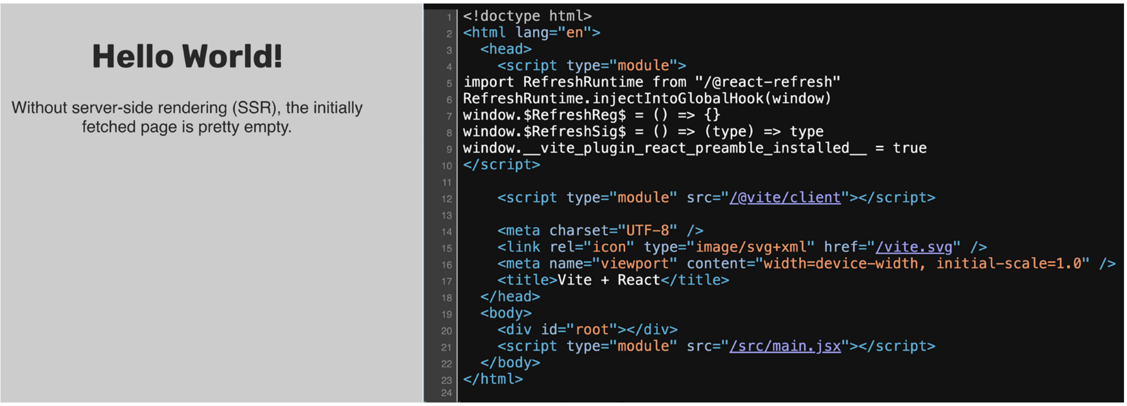
But this reliance on client-side code (and, therefore, JavaScript) also has potential disadvantages:

- If users disable JavaScript, the website will be pretty much unusable.
- The initially fetched HTML document is almost empty—data fetching and content rendering only take place after that initial HTTP request and response.

The first point might not matter too much, since only a small subset of all users will disable JavaScript and you can show an appropriate warning message via the `<noscript>` tag.

But the second point can have significant consequences. Since the initial HTML document is almost empty, users won't see any content until all the JavaScript code has been fetched and executed. While most users might not see a notable delay, depending on the device and internet connection of a user, this may take up to a few seconds for some users.

In addition, search engine crawlers (e.g., Google's crawler) will not necessarily wait for all your client-side JavaScript code to be fetched and executed when indexing your page. Therefore, those crawlers may see a mostly empty page and hence rank your website badly (or not index it at all).



Hello World!

Without server-side rendering (SSR), the initially fetched page is pretty empty.

```html
<!doctype html>
<html lang="en">
  <head>
    <script type="module">
import RefreshRuntime from "/@react-refresh"
RefreshRuntime.injectIntoGlobalHook(window)
window.$RefreshReg$ = () => {}
window.$RefreshSig$ = () => (type) => type
window.__vite_plugin_react_preamble_installed__ = true
</script>

    <script type="module" src="/@vite/client"></script>

    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>

```

Figure 15.1: The page content is nowhere to be found in the page source code (i.e., the fetched HTML document)

Figure 15.1 shows the page source code (which can be inspected by right-clicking on the website) of a typical React app. As you can see in that figure, there's almost no content between the `<body>` tags. The title (`"Hello World!"`) and the text below it are missing in that source code. The content is missing there because it's not part of the initial HTTP response. Instead, it's rendered by the transpiled React code after the page loaded (and after that code was downloaded from the server).

Of course, these disadvantages won't matter in all cases. If you're building some company-internal application, or a UI that's hidden behind some login (and hence won't be indexed anyway), or if you're only targeting users with fast devices and internet connections, you might not need to worry about these potential problems.

But if you're building a public-facing website where search engine indexing matters or that may be visited by users with slow devices or internet connections, you might want to consider getting rid of these disadvantages. And that's precisely where **SSR** can help out.

Making Sense of Server-side Rendering (SSR)

When working with React, SSR refers to the process of rendering web pages, and therefore your React components, on the server that handles the incoming HTTP request when a user visits your website.

With SSR enabled, the server will render your React component tree and hence produce the actual HTML code yielded by your components and their JSX instructions. It's this finished HTML code that's then sent back to the client. As a result, website visitors will receive an HTML file that's not empty anymore but that instead contains the actual page content. Search engine crawlers will also see that content and index the page accordingly.

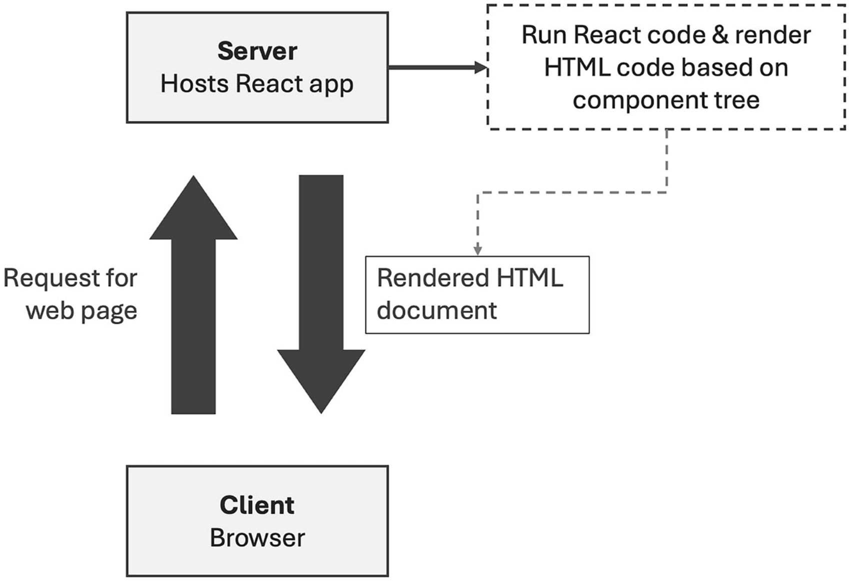


Figure 15.2: React SSR in action

Best of all, you don't lose the client-side advantages of React because, when enabling SSR, React still works on the client side as it did before! It'll take over control once that initial HTML document has been received and provide users with the same SPA experience you were able to deliver without SSR. Though, technically, when using SSR, React will be initialized slightly differently on the client. Instead of re-rendering the entire DOM there, it'll **hydrate** the page content that was rendered on the server. **Hydration** means that React will connect your component structure to the rendered HTML code (which was rendered based on that same structure, of course) and make it interactive.

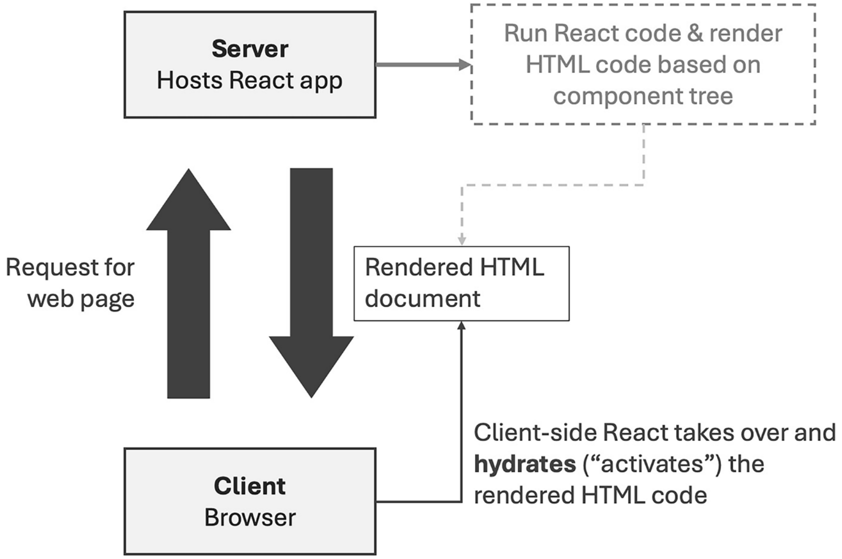

Server
Hosts React app
Run React code & render HTML code based on component tree
Request for web page
Rendered HTML document
Client
Browser
Client-side React takes over and **hydrates** ("activates") the rendered HTML code

Figure 15.3: After receiving the rendered HTML code, React hydrates the code on the client side

Consequently, you'll get the best of both worlds: non-empty, pre-rendered pages for the initial HTTP request sent by the browser, and a highly reactive web application for the user to enjoy.

Adding SSR to a React Application

It is extremely important to understand that SSR-enabled React applications need to execute code in two environments (server and browser), whereas client-side React applications only rely on the browser. Therefore, to use SSR, a server-side environment must be added to the React project—it's not enough to just adjust the React code in a few places.

For example, standard Vite-based projects don't support SSR out of the box. Consequently, if you want to support SSR, you must edit your Vite project setup (and some of your project code files) to enable executing React code on both the client and server side. For example, you must add some code that handles incoming HTTP requests and triggers React code execution on the server side.

Note

Manually enabling SSR requires backend development and build process configuration knowledge—in addition to the React knowledge you need.

Thankfully, though, as you'll learn throughout this chapter, you often don't need to go through that setup process. Instead, you can rely on frameworks like Next.js to do the heavy lifting for you.

If you're interested in manually configuring SSR in Vite-based projects, the official Vite SSR documentation is a great place to learn more: `https://vitejs.dev/guide/ssr`.

In addition, you can explore the following demo project that was set up according to the official Vite SSR instructions: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/15-ssr-next-intro/examples/02-ssr-enabled`.

The fact that manually enabling SSR is a non-trivial process that requires advanced Node.js and backend development knowledge is one of the reasons why the official React documentation recommends creating new React projects with the help of frameworks like Next.js (see `https://react.dev/learn/start-a-new-react-project`).

But it's not the only reason.

Server-side Data Fetching Is Not Trivial

Besides the non-trivial setup process, SSR-enabled projects also suffer from another possible problem: server-side data fetching is difficult.

If you're building a React app that needs to fetch data in some components (e.g., with the help of `useEffect()`, as shown in *Chapter 8*, *Handling Side Effects*), you'll find out that the data is not fetched when the component is rendered on the server. Instead, the data fetching will only occur on the client side. The server-side rendered HTML markup will not contain the content that depends on the fetched data.

The reason for this behavior is that React component functions are only executed on the server once—i.e., it's only the first component render cycle that's performed on the server. You can think of SSR producing only an initial page snapshot. Subsequent state updates are ignored, and effect functions (triggered via `useEffect()`) are also therefore not executed on the server side. As a result, data fetching that relies on effect functions and subsequent state updates will not work on the server side.

Consider this example, where a `Todos` component function uses `useEffect()` to fetch some (dummy) to-dos data from `https://jsonplaceholder.typicode.com/`:

```
import { useEffect, useState } from 'react';

import { loadTodos, saveTodo } from '../todos.js';
```

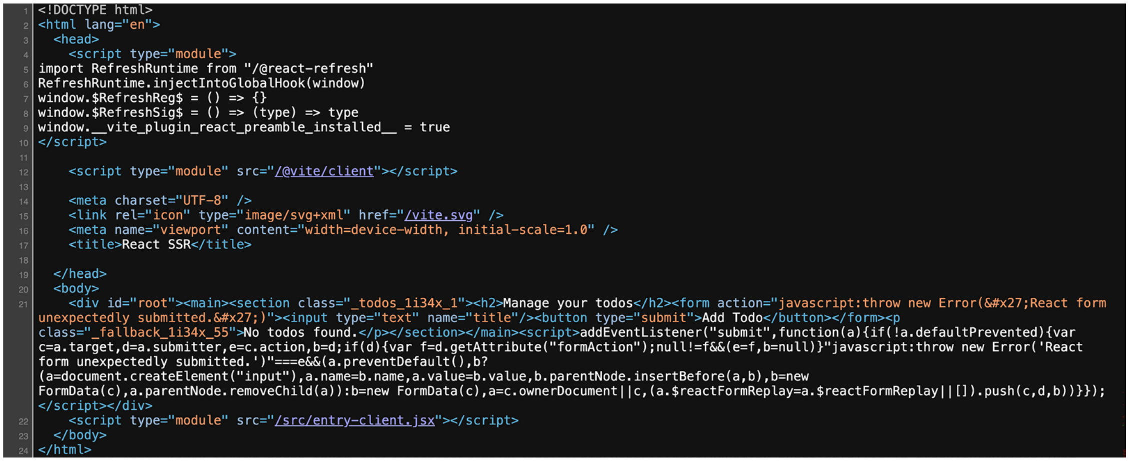
```
function Todos() {
  const [todos, setTodos] = useState();

  useEffect(() => {
    async function fetchTodos() {
      // sends HTTP request to jsonplaceholder.typicode.com
      const todos = await loadTodos();
      setTodos(todos);
    }
    fetchTodos();
  }, []);

  async function addTodoAction(fd) {
    const todo = {
      title: fd.get('title'),
    };
    const savedTodo = await saveTodo(todo);
    setTodos((prevTodos) => [savedTodo, ...prevTodos]);
  }

  return (
    <section>
      <h2>Manage your todos</h2>
      <form action={addTodoAction}>
        <input type="text" name="title" />
        <button type="submit">Add Todo</button>
      </form>
      {(!todos || todos.length === 0) && (
        <p>No todos found.</p>
      )}
      {todos && todos.length > 0 && (
        <ul>
          {todos.map((todo) => (
            <li key={todo.id}>{todo.title}</li>
          ))}
        </ul>
      )}
    </section>
  );
}
```

Note

You find the complete example code on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/15-ssr-next-intro/examples/03-ssr-data-fetching`.

When running this code on the server, there won't be any errors. Instead, the app will run as expected and fetch the dummy to-dos from the backend server.

However, the HTML document that's produced on the server will not contain the fetched to-dos. Instead, it will just contain the fallback text (`"No todos found"`).



```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <script type="module">
import RefreshRuntime from "/@react-refresh"
RefreshRuntime.injectIntoGlobalHook(window)
window.$RefreshReg$ = () => {}
window.$RefreshSig$ = () => (type) => type
window.__vite_plugin_react_preamble_installed__ = true
</script>

    <script type="module" src="/@vite/client"></script>

    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>React SSR</title>

  </head>
  <body>
    <div id="root"><main><section class="_todos_1i34x_1"><h2>Manage your todos</h2><form action="javascript:throw new Error(&#x27;React form unexpectedly submitted.&#x27;)"><input type="text" name="title"/><button type="submit">Add Todo</button></form><p class="_fallback_1i34x_55">No todos found.</p></section></main><script>addEventListener("submit",function(a){if(!a.defaultPrevented){var c=a.target,d=a.submitter,e=c.action,b=d;if(d){var f=d.getAttribute("formAction");null!=f&&(e=f,b=null)}"javascript:throw new Error('React form unexpectedly submitted.')"===e&&(a.preventDefault(),b?(a=document.createElement("input"),a.name=b.name,a.value=b.value,b.parentNode.insertBefore(a,b),b=new FormData(c),a.parentNode.removeChild(a)):b=new FormData(c),a=c.ownerDocument||c,(a.$reactFormReplay=a.$reactFormReplay||[]).push(c,d,b))}});</script></div>
    <script type="module" src="/src/entry-client.jsx"></script>
  </body>
</html>
```

Figure 15.4: The rendered HTML does not contain the actual to-dos

The generated markup does not contain the fetched to-dos because, as explained above, React component functions only execute once on the server side (and the function passed to `useEffect()` doesn't execute at all).

Due to this behavior, you can't easily perform asynchronous operations and, for example, fetch data via `useEffect()` in your React components when using SSR. Hence, the server-side rendered HTML content will never contain that data.

While you can come up with workarounds to that problem (e.g., perform the data fetching operation on the server, before executing the component functions) that's a problem that will be solved by Next.js and a concept called **React Server Components** (**RSC**).

Introducing Next.js

Next.js is a React framework—i.e., a framework that builds upon React and adds extra features and patterns to it. Specifically, Next.js adds features like file-based routing, built-in SSR, or automatic caching to improve performance. Though, most importantly, it unlocks two crucial React concepts: **React Server Components** (**RSC**) and **Server Actions**. As you will learn, these features enable server-side React code to perform asynchronous operations and, for example, fetch and render data on the server.

Thus, Next.js saves you the effort of manually enabling SSR, and, additionally, unlocks other powerful features that help with fetching data on the server.

> **Note**
>
> There are also alternative React frameworks like Remix/React Router (they were merged to bring optional fullstack React framework features to React Router) or TanStack Start.
>
> Next.js has not only existed for a very long time but it's also the most popular (measured by usage) fullstack framework at the point in time when this book was written.

This chapter will get you started with Next.js and provide a brief overview of its core concepts. The next chapter (*Chapter 16*, *React Server Components & Server Actions*) will then build upon this knowledge to dive even deeper.

Creating Next.js Projects

To use Next.js, you must first create a Next.js project. Technically, it'll still be a React project, which means you will be able to use React features like components, props, state, Hooks, or JSX. But it'll be a project that comes with the `next` package installed, and that enforces a certain folder structure that's needed by Next.js. You can't install Next.js into an existing (Vite-based) React project and start using it there. Significant adjustments to the project configuration and structure would be required. Next.js brings its own build process and does not use Vite under the hood. Hence, creating a brand-new project makes more sense.

To get started with a new Next.js project, you should run the following command in your system terminal or command prompt (in a place on your system where you want the new project folder to be created):

```
npx create-next-app@latest first-next-app
```

After running this command, you'll have to make a couple of choices in the terminal (e.g., if you want to use TypeScript).

You can confirm all those choices by simply pressing the *Enter* key, hence accepting the default option. However, you should ensure that you choose `No` for TypeScript (unless you know how to use it) and `Yes` for **App Router**. You can find a (slightly cleaned up) starting project on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/15-ssr-next-intro/examples/04-nextjs-intro`.

Inside the created project folder, a development server can be started via:

```
npm run dev
```

While the command is the same as in a Vite project, the server will actually target a different port by default. Instead of `localhost:5173` (Vite), Next.js projects use `localhost:3000` for the preview development server.

Just as in a Vite-based project, you should keep this process up and running while you're working on the project code. The underlying build process will automatically reload and update the preview website as you make changes to your code.

Note

Next.js is an established, mature framework that's never stopped innovating and changing.

In late 2022, the so-called **App Router** was introduced as a new way of structuring and building Next.js applications (the old approach is now referred to as Pages Router). This book, of course, covers the new App Router approach.

As of mid-2024 (when this edition is written), the App Router approach, despite being marked as stable, still frequently receives new features and changes.

Therefore, even though unlikely, the concepts and code explained in this book may change or break over time. The setup process described above may change, too. In such cases, a note (with instructions on how to adjust the code) will be added to the Changelog document on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/main/CHANGELOG.md`.

A newly created Next.js project comes with all its dependencies installed (`npm install` is automatically executed as part of the project creation process) and a project structure like this:

- An `app/` folder that holds route-related files (see the next section)
- A `public/` folder that can be used to store assets that should be served statically (i.e., without being changed by the build process)
- `jsconfig.json` and `nextjs.config.mjs` files for configuring the project and Next.js-specific behaviors
- `package.json` and `package-lock.json` for managing project dependencies

Hence, except for the `app/` folder, it's not too different from the structure you know from Vite. However, it is worth noting that Next.js, unlike Vite, does not enforce `.jsx` as a file extension for JavaScript files that contain JSX code. You can use it but you don't have to. For example, the starting project uses `page.js` and `layout.js`, not `page.jsx` and `layout.jsx`, even though these files contain JSX code.

Just like Vite-based projects, Next.js projects come with a build workflow that processes and transpiles your code files automatically, when running the development server or building for production (which you can do via `npm run build`).

Like pretty much all modern React project setups, Next.js projects therefore support importing style files (like `globals.css`) or images into JavaScript files. It also allows you to omit or set file extensions on imports. In addition, Next.js has CSS Modules support, too.

Put in other words: you can work in Next.js projects in pretty much the same way as you do in Vite-based projects.

Working with File-Based Routes

In Vite-based projects, you have a high degree of flexibility when it comes to the project structure. Inside the `src/` folder, you can create any subfolders and files of your choice. The names of those files and folders also don't really matter (if they're valid and use the right extensions).

When working with React Router, you would set up routes in one of your JSX code files and load any component stored in any file for any route (see *Chapter 13*, *Multipage Apps with React Router*).

In Next.js projects, that's a bit different because Next.js uses the file system for defining routes—you don't set up routes in code. As a result, while you still have lots of flexibility, there are some routing-related rules regarding the project structure and file names that must be followed—otherwise, the app will break and not work as intended.

Next.js implements file-based routing via its own built-in router. This router analyzes your file system and derives supported routes, their URL paths, and which React components to load and render based on the file and folder structure in your project.

When using the App Router approach, you therefore must store all components that should be loaded as pages inside the `app/` folder (or a nested folder) in files named `page.js`. Since all route component files must be named `page.js`, it's the parent folder names that define the route path for which the component will get loaded.

For example, you might have a file and folder structure as shown in *Figure 15.5*:

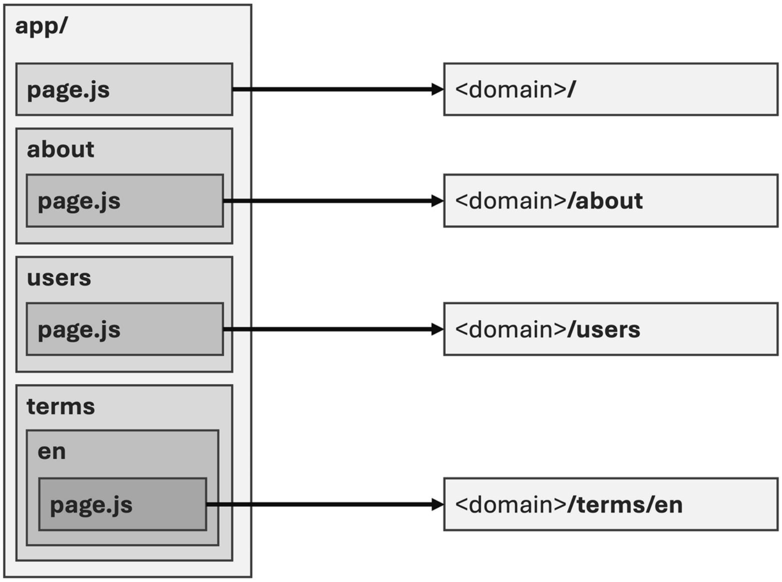


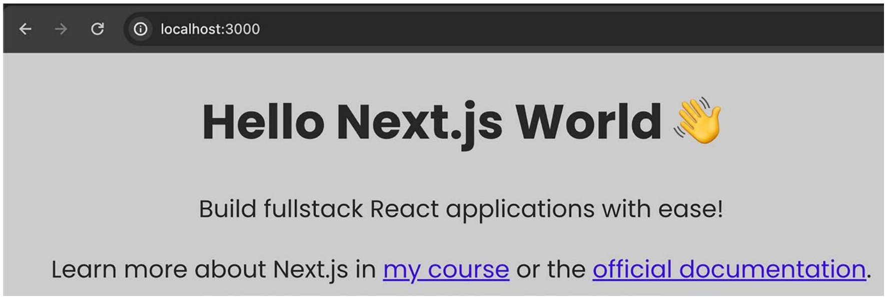
Figure 15.5: In Next.js, page.js files contain route components. The folder names determine the path

In *Figure 15.5*, you can see that four routes are defined via the file system: a root route (`/`) and the `/about`, `/users`, and `/terms/en` routes. For each route, the component stored in the respective `page.js` will be loaded and rendered onto the screen.

For example, you might have an `app/page.js` file like this:

```
export default function Home() {
  return (
    <main>
      <h1>Hello Next.js World 👋</h1>
      <p>Build fullstack React applications with ease!</p>
      <p>
        Learn more about Next.js in{' '}
        <a href="https://www.udemy.com/course/nextjs-react-the-complete-
guide/">
          my course
        </a>{' '}
        or the <a href="https://nextjs.org/">official documentation</a>.
      </p>
    </main>
  );
}
```

As you can see, a regular React component function is stored in this `page.js` file. The name of the component function does not matter—it's just important that it's a component function that's exported inside a file named `page.js`. As a result, the following content will be visible on the screen if a user visits `<domain>/` (or just `<domain>`, without the forward slash):



Hello Next.js World 👋

Build fullstack React applications with ease!

Learn more about Next.js in my course or the official documentation.

Figure 15.6: The Next.js router loads the component stored in the app/page.js file and renders its content

You can therefore easily add as many, possibly nested, routes as needed—simply by creating folders, subfolders, and `page.js` files.

Server-side Rendering with Next.js

Besides providing a built-in file-based router (and many other features that will be explored throughout this and the next chapters), Next.js has one other crucial advantage: it implements SSR out of the box. You don't have to add any files, change any configuration, or adjust any code to render React components on the server—instead, it works right from the start.

Consequently, the `app/page.js` file component (the `Home` component in the example above), is evaluated and rendered on the server side when a user visits `<domain>/`. It's the finished HTML code that's sent to the browser. And, just as with Vite-based projects with custom SSR, Next.js renders all child components that may be used inside of `page.js` on the server, too.

In addition, when building websites with Next.js, you still build React apps. That's why Next.js apps become interactive on the client side once the SSR is done. Technically, as you'll also learn in the next chapter (*React Server Components & Server Actions*), they'll be made interactive in a different way than in Vite-based SSR-enabled React apps (where React hydrates the server-side rendered markup on the client), but ultimately, your website users will have a SPA-like user experience.

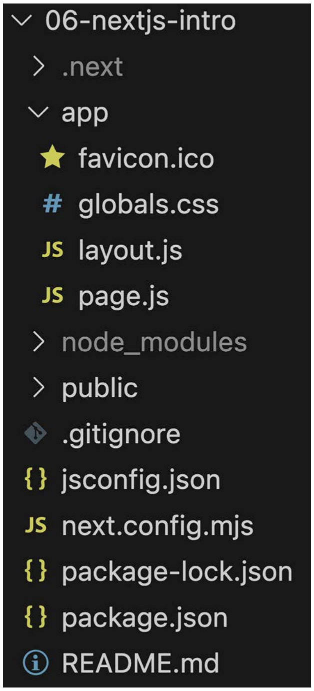
Therefore, if you want to build a React app that supports SSR, relying on a framework like Next.js instead of setting up SSR manually is recommended.

In addition, you will be able to use other helpful features, like the file-based routing system, especially since it doesn't stop at defining routes via `page.js` files. It, for example, also simplifies the process of defining layouts.

Working with Layouts

As mentioned, when it comes to routing, file names and where you store those files matter.

For example, you'll also find a `layout.js` file next to the `page.js` file in the `app/` folder from the example above.



```
06-nextjs-intro
  .next
  app
    favicon.ico
    globals.css
    layout.js
    page.js
  node_modules
  public
  .gitignore
  jsconfig.json
  next.config.mjs
  package-lock.json
  package.json
  README.md
```

Figure 15.7: Besides a page.js file, a styling file, and a favicon, a layout.js file can be found in the app/ folder

Just like `page.js`, `layout.js` is a reserved file name—i.e., that file is handled by Next.js in a special way.

This `layout.js` file also exports a component function, but the created component will not be rendered for one specific path. Instead, it is used as a wrapper around all sibling or nested pages. Thus, the `layout.js` file can be used to define JSX code that will be shared across multiple pages.

Since it's meant to be used as a wrapper component, the component function exported by `layout.js` must use the special `children` prop (see *Chapter 3*, *Components & Props*) to define the place where the wrapped page content should be displayed.

For example, you could use the `app/layout.js` file to define a global layout that adds a navigation bar above the `<main>` content:

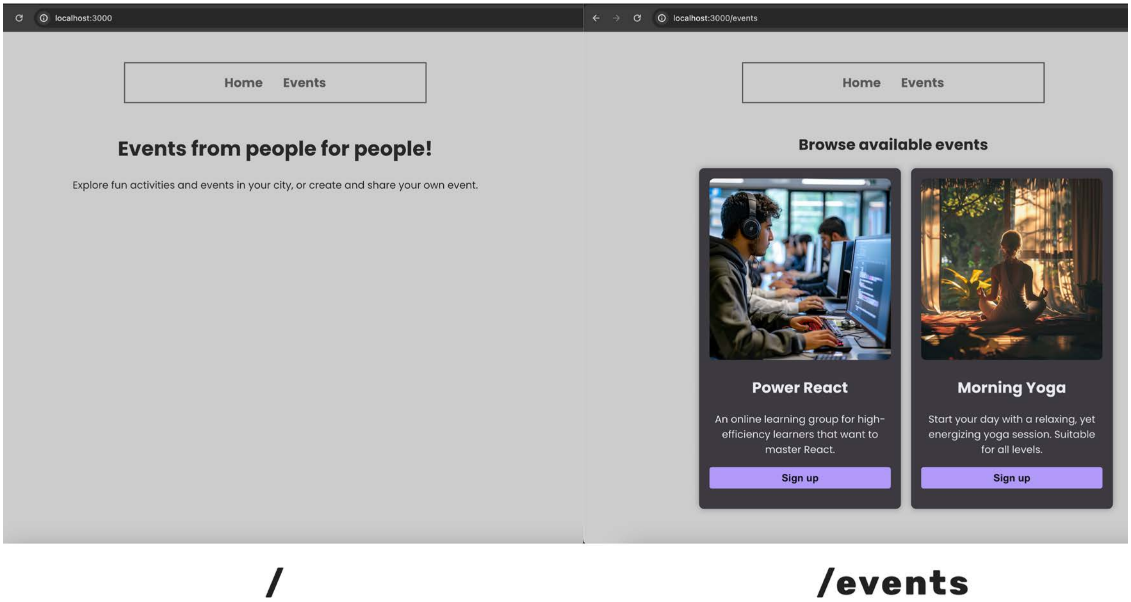
```
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>
        <header>
          <nav>
            <ul>
              <li><a href="/">Home</a></li>
              <li><a href="/events">Events</a></li>
            </ul>
          </nav>
        </header>
        <main>{children}</main>
      </body>
    </html>
  );
}
```

In this example code snippet, it's also worth noting that the `RootLayout` component renders the `<html>` and `<body>` elements. In Vite-based projects, that's not something you would do. There, you instead define a place in the `index.html` file where the rendered HTML should be injected (via the `createRoot()` function exposed by the `react-dom` package; see *Chapter 2*, *Understanding React Components & JSX*).

Next.js does not rely on such an `index.html` file; instead, it forces you to define a root `layout.js` file at the top level of the `app/` folder. It's then this root layout that must define the general structure of the rendered HTML page. However, there is no `<head>` section in that file, since Next.js will manage and inject that section behind the scenes. In addition, Next.js will also insert JavaScript and CSS imports into the rendered HTML document.

You may add more (nested) `layout.js` files if you want to have nested layouts that only wrap some of your pages. Such layouts are optional; the root layout (`app/layout.js`) is mandatory, however.

With a `layout.js` file like the one shown in the previous code example, in a project that contains an `app/page.js` file, and an `app/events/page.js` file, website users could visit both pages and see a shared navigation.



Events from people for people!

Explore fun activities and events in your city, or create and share your own event.

Browse available events

Power React

An online learning group for high-efficiency learners that want to master React.

Morning Yoga

Start your day with a relaxing, yet energizing yoga session. Suitable for all levels.

Figure 15.8: As the user navigates from/to /events, the shared header persists

In *Figure 15.8*, the main content (defined by the `page.js` files) changes but the shared navigation (set up in `layout.js`) persists.

While sharing JSX markup is the most common use case for using layouts, you can also use them to share styles by importing a CSS file into a `layout.js` file:

```
import './globals.css';

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      Unchanged JSX code…
    </html>
  );
}
```

In this and the above examples, the component function is named `RootLayout`—that name does not matter, but it must be a component that's exported.

Of course, layouts that are used to share a navigation bar become even more useful if you add working links to them…

Managing Internal Navigation

In the previous code example, the `<a>` element was used for creating links between the different Next.js application pages.

However, just like other React apps, Next.js applications become SPAs once the initial page load is done. Therefore, creating internal links via `<a>` tags is discouraged for the same reasons it was discouraged when using React Router in Vite-based React projects (compare *Chapter 13*, *Multipage Apps with React Router*).

Like React Router, Next.js (which takes care of routing in Next.js projects) provides a special `Link` component that you should use for internal links (instead of the `<a>` element):

```
import Link from 'next/link';

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>
        <header>
          <nav>
            <ul>
              <li><Link href="/">Home</Link></li>
              <li><Link href="/events">Events</Link></li>
            </ul>
          </nav>
        </header>
        <main>{children}</main>
      </body>
    </html>
  );
}
```

This `<Link>` component accepts a `href` prop, which is set to the target path. Internally, Next.js will capture link clicks and update the browser address bar and website UI accordingly by loading and rendering the required `page.js` components.

Highlighting Active Links & Using the "use client" Directive

If you want to style links differently when they lead to the currently active page, you won't find a built-in `NavLink` component as is the case with React Router. Instead, you must add your own logic by setting the `Link` component's `className` prop dynamically based on the currently active path.

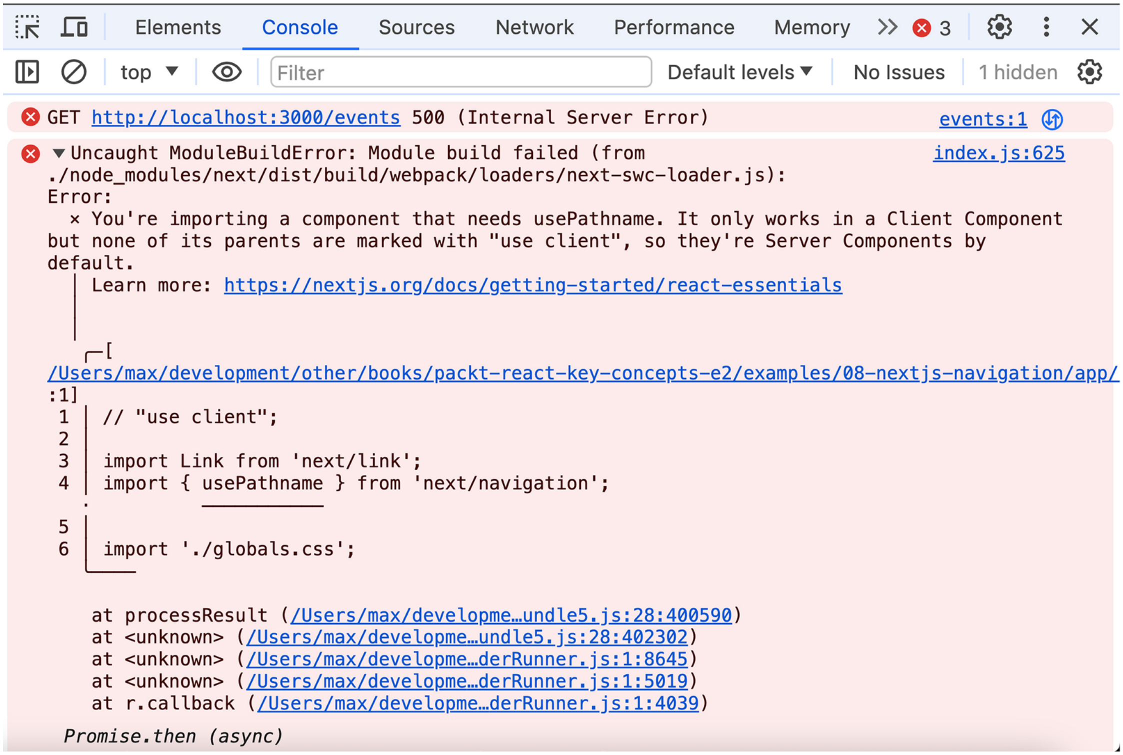
To find out which path is currently active, you can use the `usePathname()` Hook provided by Next.js:

```
import { usePathname } from 'next/navigation';

const path = usePathname();
```

For example, you could adjust the `layout.js` file to look like this:

```
import Link from 'next/link';
import { usePathname } from 'next/navigation';

import './globals.css';

export default function RootLayout({ children }) {
  const path = usePathname();
  return (
    <html lang="en">
      <body>
        <header>
          <nav>
            <ul>
              <li>
                <Link
                  href="/"
                  className={path === '/' ? 'active' : ''}>
                  Home
                </Link>
              </li>
              <li>
                <Link
                  href="/events"
                  className={path.startsWith(
                    '/events'
                  ) ? 'active' : ''}
                >
                  Events
                </Link>
              </li>
            </ul>
          </nav>
        </header>
        <main>{children}</main>
        </body>
      </html>
    );
  }
```

However, if you were to run this code, you'd get an error message:



```
GET http://localhost:3000/events 500 (Internal Server Error)                events:1
▼Uncaught ModuleBuildError: Module build failed (from                       index.js:625
./node_modules/next/dist/build/webpack/loaders/next-swc-loader.js):
Error:
  × You're importing a component that needs usePathname. It only works in a Client Component
but none of its parents are marked with "use client", so they're Server Components by
default.
  │ Learn more: https://nextjs.org/docs/getting-started/react-essentials
  │
  │
   ╭─[
/Users/max/development/other/books/packt-react-key-concepts-e2/examples/08-nextjs-navigation/app/
:1]
 1 │ // "use client";
 2 │
 3 │ import Link from 'next/link';
 4 │ import { usePathname } from 'next/navigation';
   ·          ───────────
 5 │
 6 │ import './globals.css';
   ╰────

    at processResult (/Users/max/develope…undle5.js:28:400590)
    at <unknown> (/Users/max/develope…undle5.js:28:402302)
    at <unknown> (/Users/max/develope…derRunner.js:1:8645)
    at <unknown> (/Users/max/develope…derRunner.js:1:5019)
    at r.callback (/Users/max/develope…derRunner.js:1:4039)
  Promise.then (async)
```

Figure 15.9: Next.js complains about the usage of a Hook in a Server Component

This error message sounds quite cryptic as it mentions a `Client Component` and `Server Components`. Both are crucial React concepts that will be explored in the next chapter (*React Server Components & Server Actions*).

For the current chapter, it's enough to know the fix for this problem, which is to add the `"use client"` directive at the top of the `app/layout.js` file:

```
"use client";

import Link from 'next/link';
import { usePathname } from 'next/navigation';

import './globals.css';
```

```
export default function RootLayout({ children }) {
  const path = usePathname();

  // return JSX code
}
```

`"use client"` is a so-called **directive**, i.e., an instruction that “tells” React and Next.js that this file must be handled in a special way. Adding it will get rid of the error message shown in *Figure 15.9*, thus enabling path-aware `Link` styling. As mentioned, the concrete impact of this directive will be explored in the next chapter.

Whenever you plan to use a Hook in a component in a Next.js project, the `"use client"` directive must be added—no matter if it’s a Hook provided by React or Next.js.

Note

You might wonder why `"use client"` is required for components that use Hooks. After all, this directive was not needed when using SSR in Vite-based projects.

The reason is that Next.js technically doesn’t use SSR, as introduced at the beginning of this section. Instead, Next.js (when using the App Router) uses a React feature called React Server Components. This crucial feature will be explored in great detail in the next chapter. There, you’ll also learn why exactly `"use client"` is needed in some components.

Creating & Using Regular Components

The `Link` component mentioned in the previous sections is a component offered by Next.js. But, of course, you can also build your own components—it is still a React app after all.

Besides the components that are exposed as pages (`page.js`) or layouts (`layout.js`), you can create and use component functions in any files (with any names) of your choice.

For example, you can add a `components/` folder next to the `app/` folder and add a `MainNavigation.js` file in it. This file can then hold a new `MainNavigation` component that returns the navigation-related JSX code:

```
'use client';

import Link from 'next/link';
import { usePathname } from 'next/navigation';

export default function MainNavigation() {
  const path = usePathname();

  return (
    <header>
```

```
      <nav>
        <ul>
          <li>
            <Link href="/" className={path === '/' ? 'active' : ''}>
              Home
            </Link>
          </li>
          <li>
            <Link
              href="/events"
              className={path === '/events' ? 'active' : ''}
            >
              Events
            </Link>
          </li>
        </ul>
      </nav>
    </header>
  );
}
```

Please note that `"use client"` must be added at the top of this `MainNavigation.js` file since the `usePathname()` Hook is used in the component function.

With the code moved into this newly added `MainNavigation` component, inside the `layout.js` file, `"use client"` can be removed since the `usePathname()` Hook is no longer used in that file. It's used in a child component (inside `<MainNavigation/>`) but React does not care about this.

Hence, the updated `layout.js` file looks like this:

```
import './globals.css';
import MainNavigation from '../components/MainNavigation';

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>
        <MainNavigation />
        <main>{children}</main>
      </body>
    </html>
  );
}
```

Thanks to building, outsourcing, and using the custom `MainNavigation` component, the updated `layout.js` file therefore contains a lean and concise component function again.

> **Note**
>
> With the exception of the route-related files, it is entirely up to you how you structure your Next.js project and how you name your files.
>
> As mentioned, you can store custom (non-page) components in a `components/` folder (or a folder with any other name of your choice) in any place of your choice. You can put that `components/` folder into the `app/` folder or into the root project folder.
>
> You can also not use a `components/` folder at all, and instead store components in files that are located next to your `page.js` files. Because if a file is not named `page.js`, it won't be treated as a page—so there is no danger of accidentally creating routes you don't want in your project. If you have an `app/components/MainNavigation.js` but no `app/components/page.js` file, there won't be a `/components` route. Files not named `page.js` (or one of the other reserved filenames—see the upcoming section *Other Filename Conventions*) will simply be ignored by Next.js (for routing purposes).
>
> You find more information and ideas regarding Next.js project organization in the official documentation: `https://nextjs.org/docs/app/building-your-application/routing/colocation`.

Handling Dynamic Routes

As you learned in *Chapter 13*, *Multipage Apps with React Router*, in the *From Static to Dynamic Routes* section, many React apps need to support dynamic routes, too.

For example, you might want to allow your users to visit `/events/e1` to view the details for an event with ID `e1` and `/events/e2` for an event with ID `e2` (and so on).

This is such a common requirement that Next.js, of course, supports it. You can add dynamic routes in a Next.js app by creating a folder (somewhere in the `app/` folder) that has its name wrapped by square brackets—for example, `app/events/[eventId]`. Of course, you still need a `page.js` file in that folder to actually create a route.

The part between the square brackets (`eventId`, in this example) is entirely up to you. But the square brackets tell Next.js that you're setting up a dynamic route.

The folder name between the square brackets acts as an identifier that can be used to retrieve the concrete value encoded in the URL (e.g., to retrieve `e1` in `/events/e1`).

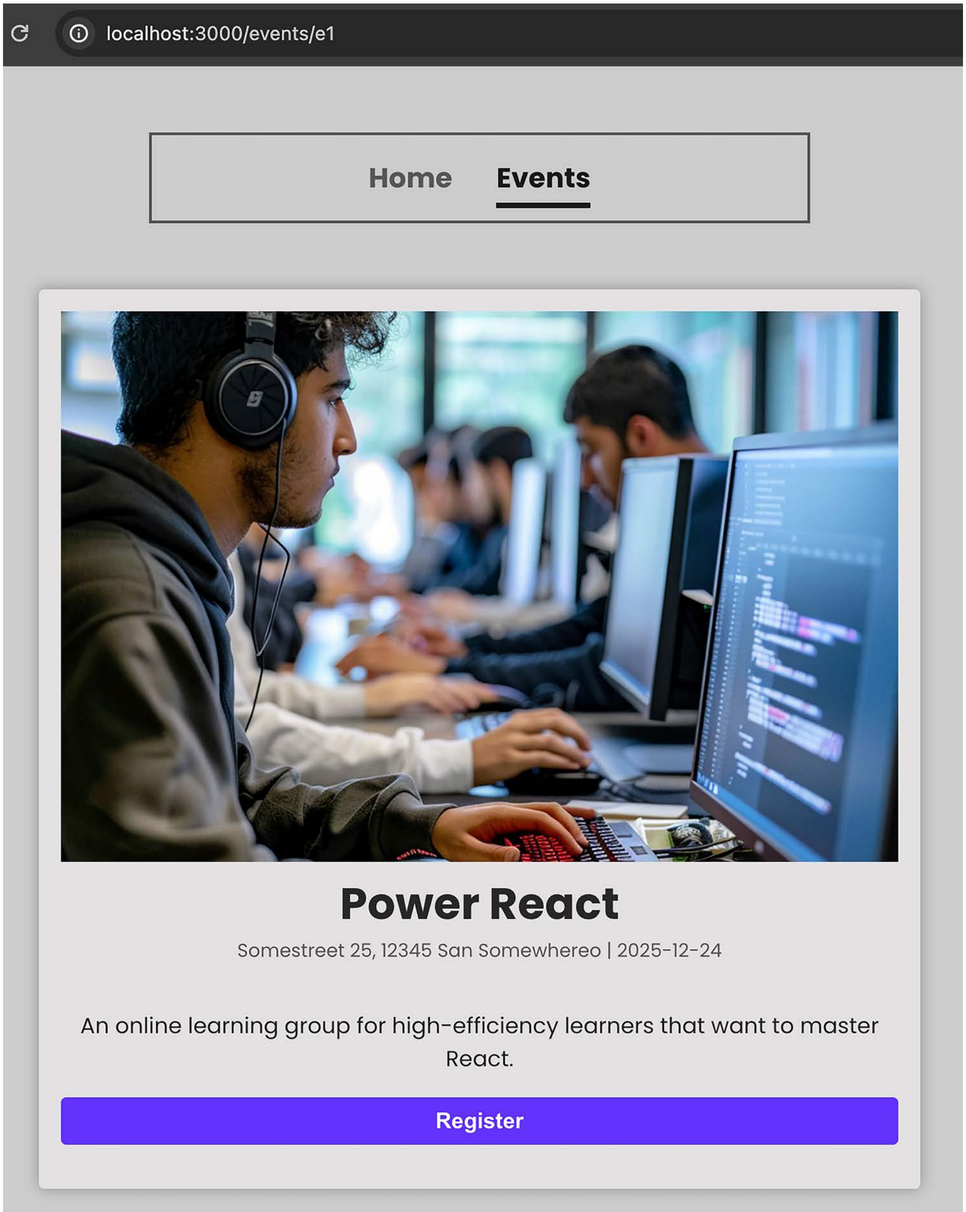
Every component that's used as a page (or layout) receives a `params` prop that's automatically set by Next.js. If it's a page or layout in a dynamic route folder or in some nested child folder, the `params` prop will hold a `Promise` which resolves to an object that contains the chosen identifiers (like `eventId`) as keys and the concrete URL path values (like `e1`) as values for those keys. Since `params` holds a `Promise`, `await` must be used on it to get access to the underlying object.

For example, the `app/events/[eventId]/page.js` file would ensure that the component exported inside the `page.js` file gets rendered for visits to `/events/e1`, `/events/e2`, etc. This page component can then output event details with the help of the following code:

```
// getEventById is a custom dummy function to load event data
import { getEventById } from '@/lib/events';

export default async function EventDetailsPage({ params }) {
  // params.eventId exists because of folder name => [eventId]
  const { eventId } = await params;
  const event = getEventById(eventId);

  return (
    <div id="event-details">
      <header>
        <img src={`/${event.image}`} alt={event.title} />
        <h1>{event.title}</h1>
        <p>
          {event.location} | {event.date}
        </p>
      </header>
      <p>{event.description}</p>
      <p>
        <button>Register</button>
      </p>
    </div>
  );
}
```

In this example, the automatically provided `params` prop is used to get access to the `eventId` encoded in the URL. If some other identifier than `eventId` would be used in the folder name, that alternative name would be used to access the path value (e.g., for `[id]/page.js`, you'd access `(await params).id`).

As a result, users can visit this dynamic route and explore the event details for a chosen event ID.



Power React

Somestreet 25, 12345 San Somewhereo | 2025-12-24

An online learning group for high-efficiency learners that want to master React.

Figure 15.10: The event details are loaded and displayed for /events/e1

Note

You can find the complete example code, including the `lib/events.js` file, on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/15-ssr-next-intro/examples/08-nextjs-dynamic-routes`.

Of course, when working with dynamic routes, you typically also need links to those dynamic paths in some parts of your application. Therefore, in this example, the `app/events/page.js` file contains code that dynamically renders a list of event items, where every item has a link to its detail page:

```
import Link from 'next/link';

import { getEvents } from '@/lib/events';

export default function EventsPage() {
  const events = getEvents();
  return (
    <div id="events">
      <h2>Browse available events</h2>
      <ul>
        {events.map((event) => (
          <li key={event.id}>
            <img src={event.image} alt={event.title} />
            <div>
              <h2>{event.title}</h2>
              <p>{event.description}</p>
              <p>
                <Link
                  href={`/events/${event.id}`}>Explore Event</Link>
              </p>
            </div>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

Clicking these links will take users to the event detail page for the specific event ID.

Note

Static routes, dynamic routes, and nested routes are the most important route types you need to know when working with Next.js. You'll use them for most of your routes.

In addition, Next.js also offers other (more advanced and niche) route types and features that are worth exploring if you decide to dive deeper into Next.js: `https://nextjs.org/docs/app/building-your-application/routing`.

Besides different route types that are enabled by using the proper folder names, Next.js also offers additional reserved filenames.

Other Filename Conventions

Next.js does not just offer a variety of route types and routing-related features—it also offers more reserved filenames than just `page.js` and `layout.js`.

Therefore, when working with the Next.js App Router, you should also be aware that the following reserved filenames exist, too:

- `loading.js` files can be added next to or above `page.js` and `layout.js` files to define components that should be displayed whilst the page (or layout) component is fetching data.
- `error.js` files can be added in the same places as `loading.js` files to render error fallback components in case one of the sibling or child pages throws an error.
- `not-found.js` files can be added to display fallback content in case a website visitor tries to load a non-existent route or resource.
- `route.js` files can be added to set up routes that do not render components but instead return data (e.g., in the JSON format).

You can learn more about these file types and even more file name conventions in the official documentation: `https://nextjs.org/docs/app/building-your-application/routing#file-conventions`.

You'll also see some of these file types in action in the next chapter.

Diving Deeper into Next.js

At this point, you have a solid Next.js foundation but, as mentioned in the previous section, you can dive deeper into Next.js with the help of the official documentation.

There, besides learning more about routing, route types, and filenames, you can also explore how Next.js helps with caching, styling, or managing page metadata. Since this book is primarily about React itself, and not about Next.js, covering all these topics here would quickly bloat this book.

That's why this chapter focused on setting a solid React SSR and Next.js foundation. The essentials covered throughout this chapter will help with understanding more advanced React and Next.js features like React Server Components in the next chapter. In addition, thanks to these fundamentals, you'll also be able to quickly learn more about Next.js with the help of the official documentation or dedicated Next.js books or courses.

Summary and Key Takeaways

- By default, Vite-based React apps (like most React apps that don't use Next.js or a similar framework) only support client-side rendering.
- Without SSR, a relatively empty `index.html` file is sent to the client.

- This can lead to bad user experiences (if users see an empty page for a prolonged period) or suboptimal search engine ranking.
- You can enable SSR by manually adjusting React projects (code and build process) to support component function execution on the server side.
- To avoid custom SSR setup work and take advantage of many other benefits, you can use frameworks like Next.js.
- Next.js projects come with built-in SSR support and can be created via the `npx create-next-app` command.
- Modern Next.js uses the App Router approach, which takes advantage of an `app/` directory that is used for setting up routes with the help of the file system.
- Inside `app/`, you define pages by creating folders that contain `page.js` files (e.g., `app/about/page.js` adds support for an `/about` route).
- To share JSX code (and logic or styles) across pages, you can add `layout.js` files.
- Next.js also offers other reserved filenames to handle fallback content that's shown while loading data or to handle errors.
- You can link between pages via Next.js' `Link` component.
- When using React Hooks (like `useState()` or Next.js' `useRouter()`), you must add the `"use client"` directive at the top of the file that uses the Hook.
- Besides static pages (like `app/events/page.js` or `app/about/page.js`), you can also set up dynamic pages by enclosing a folder name with square brackets (e.g., `app/events/[eventId]/page.js`).
- Dynamic path parameter values can be extracted in the loaded page component by using the special `params` prop that's set on the component by Next.js.
- Asynchronous operations can be problematic when using SSR—or, at least, they can't be executed in components that are rendered on the server, hence forcing the client-side code to perform them. At least when not using React Server Components.

What's Next?

At this point, you have learned a lot about SSR in React apps and about Next.js. You're able to create Next.js projects, define routes, render page components, add navigation, and work with dynamic paths.

You also learned that Next.js comes with built-in SSR. Thus, all React components (built-in and custom, page and non-page) are rendered on the server when a website visitor sends a request.

Modern Next.js does not stop there, though—instead, unlike the custom SSR setup introduced at the beginning of this chapter, Next.js projects that use the App Router help with asynchronous data fetching on the server side by unlocking React's **React Server Component** feature. It's that feature, and **Server Actions**, that will be explored in great detail in the next chapter!

Test Your Knowledge!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to examples that can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/15-ssr-next-intro/exercises/questions-answers.md`:

1. Which two main advantages can SSR offer?
2. What are some potential disadvantages or weaknesses of SSR?
3. How does Next.js help with SSR?
4. How are routes configured in Next.js (when using the "App Router")?
5. What's special about a page component in Next.js?
6. What's the purpose of layout components in Next.js?
7. Where can you store non-page (and non-layout) React components in a Next.js project?
8. When and where do you need to add the `"use client"` directive?

Apply What You Learned

With all the newly gained knowledge about Next.js, it's time to apply it to a real demo project—a demo application that will be rendered on the server.

In the following section, you'll find an activity that allows you to practice working with Next.js. As always, you will also need to employ some of the concepts covered in earlier chapters.

Activity 15.1: Migrating a Vite-Based React Router App

In this activity, your job is to build upon the Vite-based app from *Activity 13.1*. That app was built with Vite and React Router. Your job is to migrate it from Vite and React Router to Next.js.

Therefore, you should create a new Next.js project (using the App Router) and rebuild the same app in that project.

Note

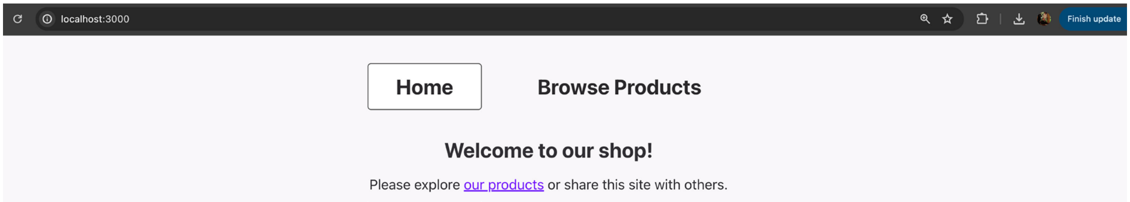
You can find the starting code for this activity at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/15-ssr-next-intro/activities/practice-1-start`. When downloading this code, you'll always download the entire repository. Make sure to then navigate to the subfolder with the starting code (`activities/practice-1-start`, in this case) to use the right code snapshot.

Since your task is to migrate the project that was built in *Activity 13.1*, you might also want to use the finished code from that activity. You can find it here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/13-routing/activities/practice-1`.

After downloading the code and running `npm install` in the project folder to install all required dependencies, the solution steps are as follows:

1. If you created a new Next.js project (i.e., if you're not using the provided starting snapshot), clean up the `layout.js` and `page.js` files to remove everything but the component functions.
2. Create two new routes: a `/products` route and a `/products/<some-id>` route.
3. Migrate the `data.js` file into the Next.js project (e.g., into a `lib/` folder in the root project folder).
4. Update the page components to load and display the data provided by the `data.js` file.
5. Create a new `components/` folder and migrate (copy) the `MainNavigation` component into this folder.
6. Update the `MainNavigation` component (and any other component that needs it) to use Next.js' `Link` component.
7. Highlight active links with the help of the `usePathname()` Hook—don't forget about the `"use client"` directive!
8. Migrate the styles from the `index.css` file into the `globals.css` file. Make sure that the file gets imported into the root layout file.

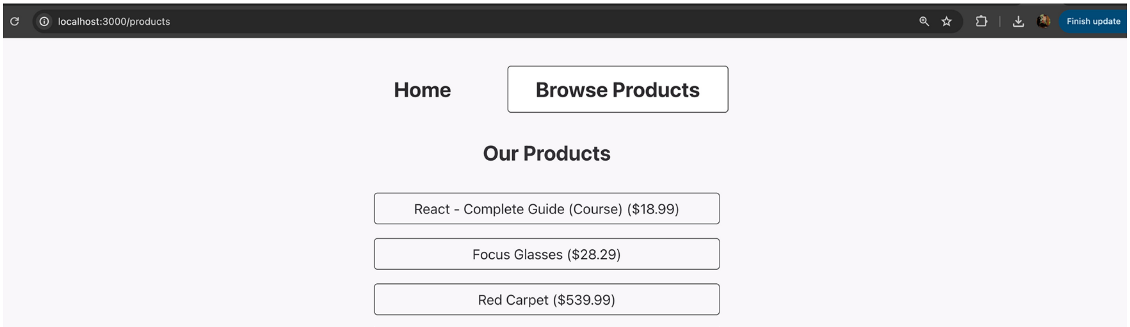
The expected result should look as shown in the following screenshots:



Welcome to our shop!

Please explore our products or share this site with others.

Figure 15.11: The home page content



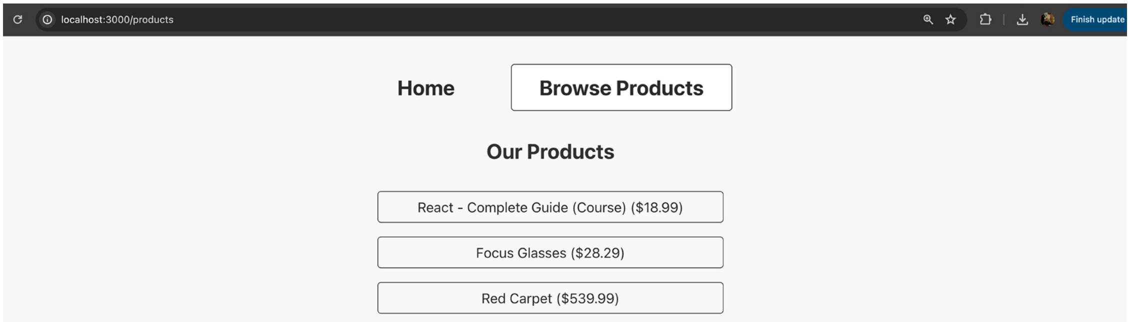
Our Products

React - Complete Guide (Course) ($18.99)

Focus Glasses ($28.29)

Red Carpet ($539.99)

Figure 15.12: The /products page content


localhost:3000/products
Finish update
Home
Browse Products

Our Products

React - Complete Guide (Course) ($18.99)

Focus Glasses ($28.29)

Red Carpet ($539.99)

Figure 15.13: The /products/<some-id> page content

Note

You will find the full code for this activity, and an example solution, here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/15-ssr-next-intro/activities/practice-1`.

16

React Server Components & Server Actions

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Create and use **React Server Components** (**RSCs**)
- Describe how (and when) RSCs are executed and rendered to the screen
- Fetch data and perform asynchronous operations with the help of RSCs
- Draw server-client boundaries by building and using client components
- Perform server-side data mutations with the help of Server Actions
- Update the **user interface** (**UI**) in response to Server Actions

Introduction

In the previous chapter, you learned that you can use **server-side rendering** (**SSR**) to render React components on the server. SSR ensures that users receive a fully populated HTML document upon their initial HTTP request, not an almost empty page shell. You were also introduced to Next.js and learned how you may use that framework to build React apps that come with SSR (and many other useful features) out of the box.

This chapter builds upon the previous one—specifically, you'll learn about two crucial React features that are unlocked by Next.js: **React Server Components** (**RSCs**) and **Server Actions**.

Throughout this chapter, you'll learn how these two features help with data fetching and mutations, and why you can't use them in every React project, even though they're technically part of React—not Next.js.

Note

RSCs and Server Actions are relatively new React features. Supporting them in custom React projects is tricky, as you will learn throughout this chapter.

Whilst unlikely, it is possible that concepts or features related to RSCs or Server Actions change. It's also possible that supporting these features in custom projects gets easier.

That's why this book comes with a dedicated document that tracks any significant changes you should be aware of: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/main/CHANGELOG.md`.

The Problem with Server-side Data Fetching

If you have an SSR-enabled React app, either by manually enabling it, for example, in a Vite-based project, or by using a framework like Next.js, your React component functions get executed on the server. Thus, any data required by those components should be fetched on the server.

But as explained in the previous chapter, in the *Server-side Data Fetching Is Not Trivial* section, sending HTTP requests with the help of `useEffect()` or trying to update the UI via `useState()` does not work when using SSR. On the server, React only calls the component functions once—it does not re-execute them when the state changes. It also doesn't call your effect functions.

This is a serious limitation since many React apps need to fetch data from some backend or a database. Not being able to fetch and render that data on the server means that website visitors will again receive incomplete HTML documents (and wait for the data to be fetched on the client side), and search engine crawlers will not see the most important content of the web page.

That's one of the reasons why React introduced **RSCs**.

Introducing RSCs

RSCs, despite their name, are not necessarily components that run on a server. Instead, their defining characteristic is that their component functions are never, under any circumstances, executed on the client side!

Consequently, RSCs may be executed on a server, but they may also be called during the build process, hence pre-generating components at build time. They definitely won't be executed in the browser, though.

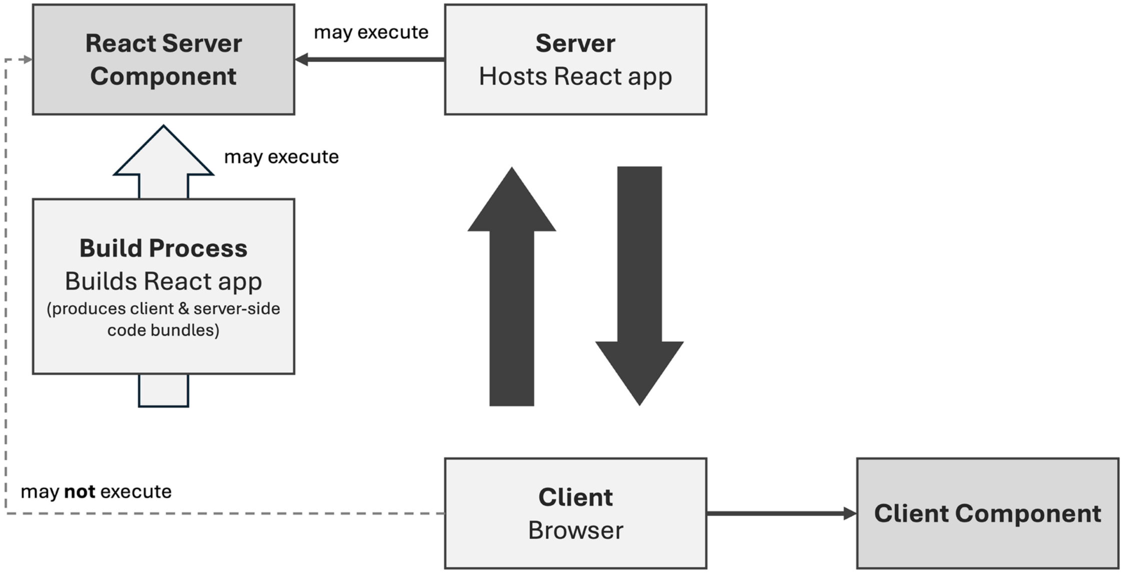

React Server Component
may execute
Server
Hosts React app
may execute
Build Process
Builds React app
(produces client & server-side code bundles)
may not execute
Client
Browser
Client Component

Figure 16.1: RSCs can't be called from the client side

But what's the purpose of RSCs? How are they created and used?

Making Sense of RSCs

The core idea behind RSCs is that you can build components that render outside of the browser (e.g., on the server). As a result, these components can execute code that wouldn't work in the browser—for example, because Node.js-specific APIs are used, or code that relies on credentials (e.g., database credentials) that must not be exposed to the client.

Unlike "normal" components (client components) that are rendered via SSR, RSCs can be rendered (on the server) after the initial page load. Hence, RSCs are not just about rendering an initial page snapshot. In addition, RSCs can fetch data on the server side. Later in this chapter, the *RSCs vs Server-side Rendering* section will take a closer look at the relationship between RSCs and "normal" components rendered via SSR.

Hence, RSCs solve an important problem: they allow you to intertwine frontend and backend React code. Whereas, in the past, before RSCs, you typically had to build separate backend and frontend web applications, you can now build integrated fullstack apps that blend server-side and client-side React code.

Using RSCs therefore offers various advantages:

- Building fully integrated fullstack applications where the backend and frontend are closely connected and use the same server becomes much easier.
- Asynchronous server-side data fetching inside of components becomes possible: Unlike on the client side (or when using SSR), React allows you to use `async/await` and return a `Promise` value in your component functions.
- Website visitors download smaller client-side JavaScript bundles since the code of RSCs is omitted.
- Running compute-heavy operations or using large third-party libraries gets easier since the operations and their code can be outsourced to the server (or to the build process).
- Code or credentials that shouldn't be accessible by your website users can be moved into RSCs.

For example, thanks to RSCs, you can create components like this:

```
import pg from 'pg'; // pg package (more info: node-postgres.com)
const { Client } = pg

const client = new Client({
  user: 'username',
  password: 'your-password',
  host: 'my.database-server.com',
  port: 5334,
  database: 'demo',
});

async function ProductsPage() {
  await client.connect();
  const res = await client.query('SELECT * FROM products');
  await client.end();

  return (
    <ul>
      {res.rows.map(row => <li key={row.id}>{row.title}</li>)}
    </ul>
  );
}
```

The `ProductsPage` component contains code that reaches out to a PostgreSQL database to fetch product data from there.

Without RSCs, this kind of component would be impossible to build and use. You wouldn't be allowed to use `async/await`, the `pg` package might rely on some APIs that are not available in the browser, and you would expose your database credentials in the client-side code bundle.

All these things are allowed when building RSCs. React explicitly does allow you to return a `Promise` (and hence use `async/await`) when building an RSC. Since the code is guaranteed to never end up on the client side, connecting to a database is safe, too.

Therefore, you can easily build fully integrated fullstack apps where backend and frontend code blend seamlessly.

However, using RSCs is both simple and complex at the same time, as the next section will explain.

Creating & Using RSCs

In a Next.js project that uses the App Router, all React components, no matter if used as pages or nested in some other component, are, by default, RSCs.

As you can tell if you inspect any React component function in a Next.js project, there really is nothing special about them. They look like regular React components:

```
export default function ServerComponentInfo() {
  return <p>This is a React Server Component.</p>;
}
```

You may use `async/await` with them, but you don't have to. You may use server-side APIs and packages, but you don't have to. So, creating RSCs is simple—they're just normal components after all.

The same is true for using them—you use them as you always used React components: as custom JSX elements:

```
<ServerComponentInfo />
```

As you can see, you wouldn't be able to tell that this is a special kind of component. It's created and used as you learned it through this entire book.

Nonetheless, all the other components from all the other chapters of this book, which were used in Vite-based React projects, were not RSCs. They were regular components or **client components**.

So, what makes the components in a Next.js project special? Why is a feature provided by React available in Next.js projects but not necessarily in other React projects (e.g., in Vite-based projects)?

Unlocking RSCs in React Projects

RSCs are a feature provided by React, not Next.js. Yet, not all React projects can use this feature.

The reason for that, and for why RSCs are available in Next.js projects, is the Next.js build process and what Next.js does to these components (and to the entire React project code, actually) behind the scenes. At a high level, you can think of Next.js doing the following things:

- The build workflow and bundling process separates server and client components to ensure that no RSC code ends up on the client side.
- Next.js sets up API endpoints (i.e., URL addresses to which the client-side code may send requests) that trigger RSC component functions on the server and return instructions that allow the client-side React code to update the UI.

- Next.js calls these endpoints when needed—for example, when navigating to a new page.
- Next.js passes the API response (which contains these rendering instructions) to React, which uses the returned instructions to update the UI as needed.

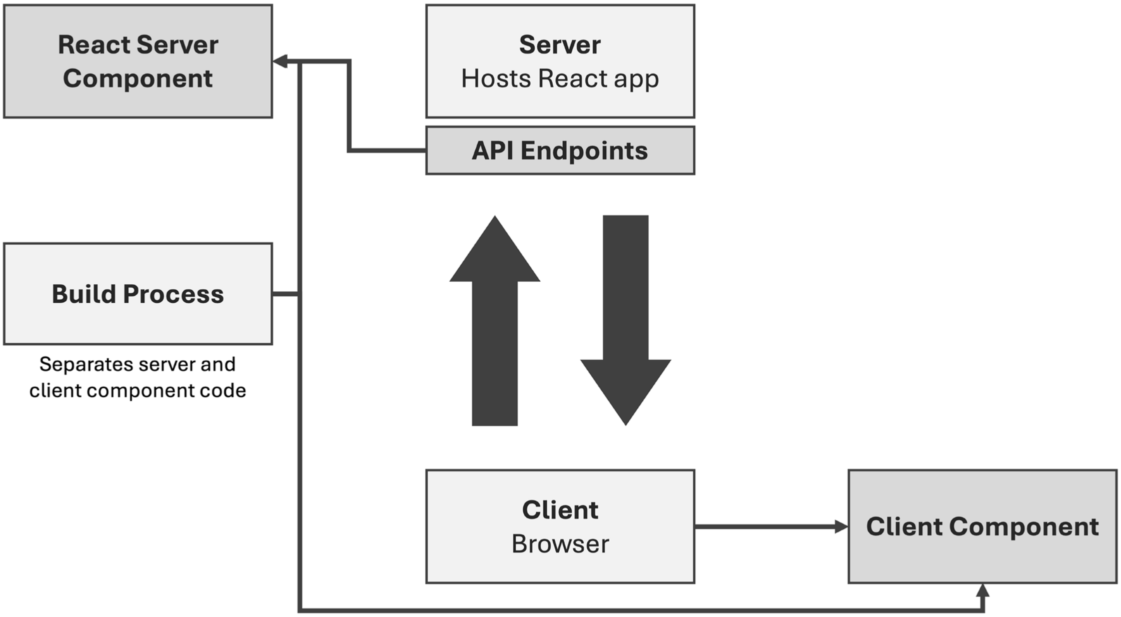

React Server Component
Server
Hosts React app
API Endpoints
Build Process
Separates server and client component code
Client
Browser
Client Component

Figure 16.2: Client and server component code are separated; communication happens via HTTP requests

Technically, it's a bit more complex than that, but for the purpose of this book and for using the feature, a deep understanding of the internals is not required—just as you don't need to understand what exactly happens internally when using `useState()`, for example.

You can verify the mentioned points by running a demo Next.js project you find here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/16-rsc-server-actions/examples/01-rsc-intro`.

This demo app consists of two basic page component files: `app/page.js` and `app/info/page.js`. The main page component (the `Home` component inside `app/page.js`) outputs a `ServerComponentInfo` component:

```
import ServerComponentInfo from '../components/ServerComponentInfo';

export default function Home() {
  return <ServerComponentInfo />;
}
```

That component in turn simply outputs some static, hardcoded content:

```
import Link from 'next/link';
```

```
export default function ServerComponentInfo() {
  return (
    <div id="rsc-info">
      <p>This is a React Server Component.</p>
      <p><Link href="/info">Learn More</Link></p>
    </div>
  );
}
```

Both the `Home` and the `ServerComponentInfo` components are RSCs—simply because they are components in a Next.js project. As mentioned earlier, all components in Next.js projects are server components by default. If these components were part of a Vite-based React project that is not set up to support RSCs, these components would instead be "normal" components (client components).

In the same demo project, there is also a component for the `/info` page. This component contains some code that wouldn't work in a component that isn't an RSC:

```
import fs from 'node:fs/promises';

export default async function InfoPage() {
  const info = await fs.readFile('data/rsc-info.json', 'utf-8');
  const { summary } = JSON.parse(info);

  return (
    <div id="info-page">
      <h1>Understanding React Server Components</h1>
      <p>
        {summary}
      </p>
    </div>
  );
}
```

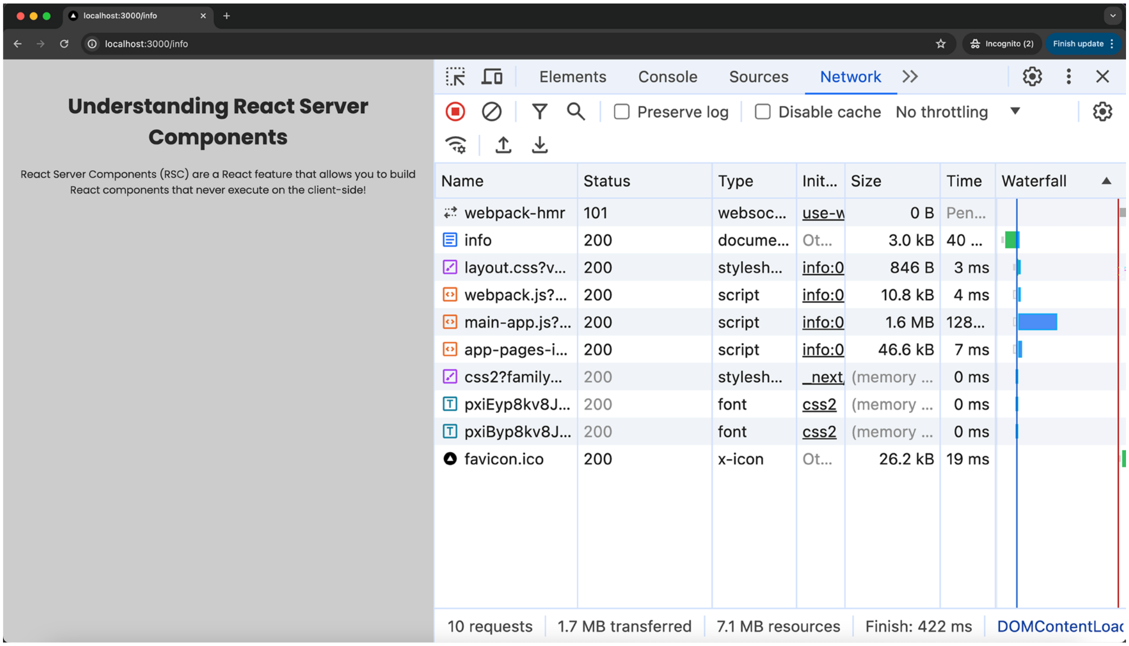
This code wouldn't work in any of the (Vite-based) React projects you saw before in this book because of the following reasons:

- The `InfoPage` component uses Node's `fs` package to load data from a `rsc-info.json` file (which is part of the project).
- The component uses `async/await`, hence returning a `Promise` that eventually yields the JSX code (i.e., the React elements).

In projects that do not support RSCs, you're not able to use server-side APIs since all the code runs in the browser. You're also not allowed to return a `Promise` in your components. In non-RSCs, that would not be considered a valid component function return value. When working with RSCs, both things are allowed and possible, though.

As mentioned in the *Making Sense of RSCs* section, using server-side functionalities (like Node.js APIs) is something that's unlocked because the `InfoPage` component, like all components in Next.js projects, is an RSC. For RSCs, React also supports the usage of `async/await`.

Consequently, as expected, you won't find the code of the `InfoPage` component in the client-side JavaScript code bundles. You can verify this by visiting the `/info` page. If you open the **Network** tab in the browser developer tools, and you then reload the page, you'll see all the HTTP requests sent to the server. This includes all requests for JavaScript code files that are needed on the client side of this React app.



Understanding React Server Components

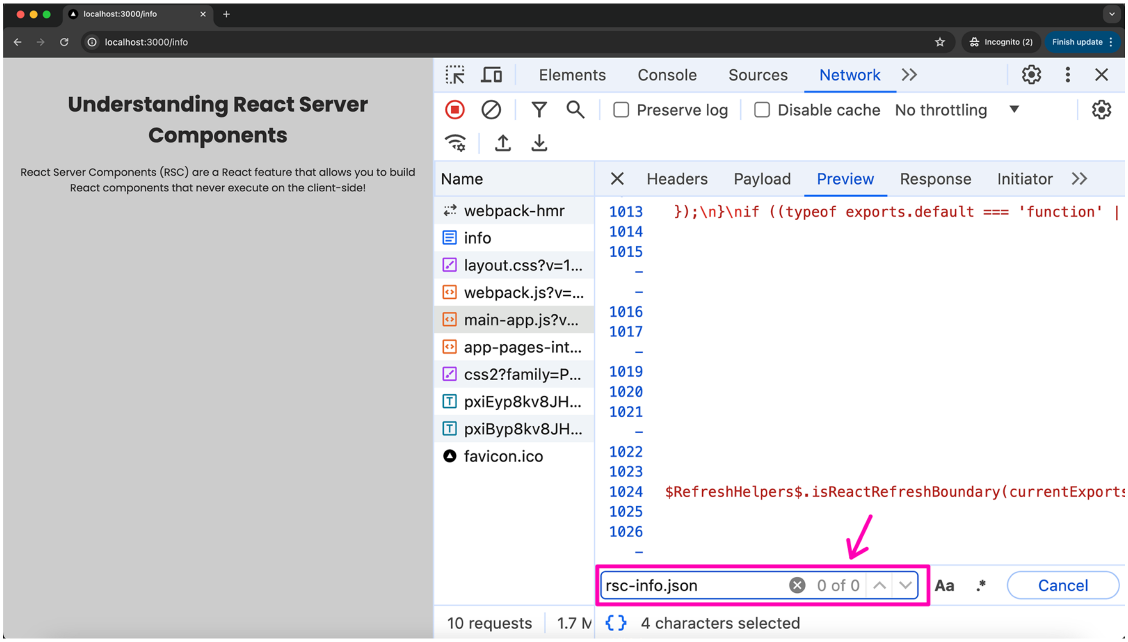
React Server Components (RSC) are a React feature that allows you to build React components that never execute on the client-side!

Name	Status	Type	Init...	Size	Time	Waterfall
webpack-hmr	101	websoc...	use-w	0 B	Pen...	
info	200	docume...	Ot...	3.0 kB	40 ...	
layout.css?v...	200	stylesh...	info:0	846 B	3 ms	
webpack.js?...	200	script	info:0	10.8 kB	4 ms	
main-app.js?...	200	script	info:0	1.6 MB	128...	
app-pages-i...	200	script	info:0	46.6 kB	7 ms	
css2?family...	200	stylesh...	_next	(memory ...	0 ms	
pxiEyp8kv8J...	200	font	css2	(memory ...	0 ms	
pxiByp8kv8J...	200	font	css2	(memory ...	0 ms	
favicon.ico	200	x-icon	Ot...	26.2 kB	19 ms	

10 requests | 1.7 MB transferred | 7.1 MB resources | Finish: 422 ms | DOMContentLoac

Figure 16.3: When visiting /info, requests for CSS, JS, and some other files are sent to the server

If you then go through all the JavaScript files requested and search for `rsc-info.json` in the downloaded code files, you won't have any matches in any file. This proves that this code, which is part of the `InfoPage` component function, does not end up in any client-side code bundle.



Understanding React Server Components

React Server Components (RSC) are a React feature that allows you to build React components that never execute on the client-side!

```
});\n}\nif ((typeof exports.default === 'function' |
$RefreshHelpers$.isReactRefreshBoundary(currentExport
```

Search: rsc-info.json — 0 of 0

10 requests

4 characters selected

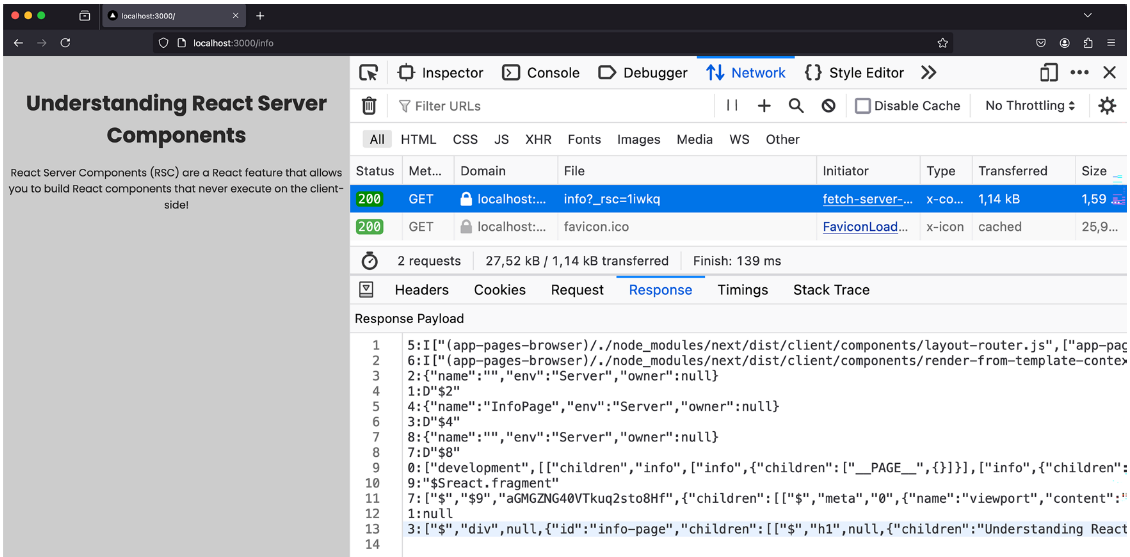
localhost:3000/info
Incognito (2)
Finish update
Elements
Console
Sources
Network
Preserve log
Disable cache
No throttling
Name
Headers
Payload
Preview
Response
Initiator
webpack-hmr
info
layout.css?v=1...
webpack.js?v=...
main-app.js?v...
app-pages-int...
css2?family=P...
pxiEyp8kv8JH...
pxiByp8kv8JH...
favicon.ico
Aa
Cancel
1.7 M

Figure 16.4: The data source filename that's included in the RSC code can't be found on the client side

How does the content fetched from the `rsc-info.json` file show up on the screen then?

This gets answered if you use a different browser than Chrome or Edge. This is required since there is a bug with the **Network** tab in the developer tools of Chrome/Edge that leads to the response of a request being hidden under certain circumstances.

Instead, you can, for example, use Firefox, to visit the root page (`/`). There, click the link that's visible on the page to navigate to the `/info` page. As you do so, one new HTTP request will be sent. If you inspect that request and its response (in Firefox's browser developer tools), you'll see the serialized RSC instructions that are returned by the server.



Understanding React Server Components

React Server Components (RSC) are a React feature that allows you to build React components that never execute on the client-side!

Status	Met...	Domain	File	Initiator	Type	Transferred	Size
200	GET	localhost:...	info?_rsc=1iwkq	fetch-server-...	x-co...	1,14 kB	1,59
200	GET	localhost:...	favicon.ico	FaviconLoad...	x-icon	cached	25,9...

2 requests | 27,52 kB / 1,14 kB transferred | Finish: 139 ms

```
5:I["(app-pages-browser)/./node_modules/next/dist/client/components/layout-router.js",["app-pag
6:I["(app-pages-browser)/./node_modules/next/dist/client/components/render-from-template-contex
2:{"name":"","env":"Server","owner":null}
1:D"$2"
4:{"name":"InfoPage","env":"Server","owner":null}
3:D"$4"
8:{"name":"","env":"Server","owner":null}
7:D"$8"
0:["development",[["children","info",["info",{"children":["__PAGE__",{}]}],["info",{"children":
9:"$Sreact.fragment"
7:["$","$9","aGMGZNG40VTkuq2sto8Hf",{"children":[["$","meta","0",{"name":"viewport","content":
1:null
3:["$","div",null,{"id":"info-page","children":[["$","h1",null,{"children":"Understanding React
```

Figure 16.5: Serialized instructions for client-side React are received from the server

As you can see, it's not HTML content that's received as a response. Instead, it's a bunch of serialized instructions that are translated to DOM elements by React on the client side (i.e., in the browser).

Therefore, as you can tell, building and using RSCs is simple, but preparing the project to handle them is not. Instead, you need a build process that separates client and server code, and API endpoints that invoke server component functions on the server. You also need client-side code that sends requests to those API endpoints whenever the server components should be rendered.

RSCs and Server Actions Can't Be Used in All Projects

Thus far in this book, whenever some new React feature was introduced, you could simply use it in your React project, no matter whether it was a project created and managed by Vite or any other tool (e.g., **create-react-app**).

With RSCs and Server Actions, this changes. Due to the many things that must be done behind the scenes (see the previous section), even though these are features provided by React, you can't just start using them in any React project. Instead, to unlock these features, you must have a project that's configured to support them.

As a result, at the point of time where this book is written, RSCs and Server Actions can really only be used with the help of frameworks that integrate and actively support these features—for example, the Next.js framework.

Of course, it is technically possible to set up a project that supports both features on your own, but it requires advanced knowledge regarding backend development and build workflow configuration. Consequently, most React projects that need these features rely on frameworks like Next.js. Since the way you work with RSCs and Server Actions will always be the same, no matter in which kind of project you use them, this book will therefore ignore the custom setup part and instead focus on how to use these two core concepts.

RSCs vs Server-side Rendering

At first sight, using RSCs may look similar to SSR React components. After all, both concepts are about running some code outside of the browser.

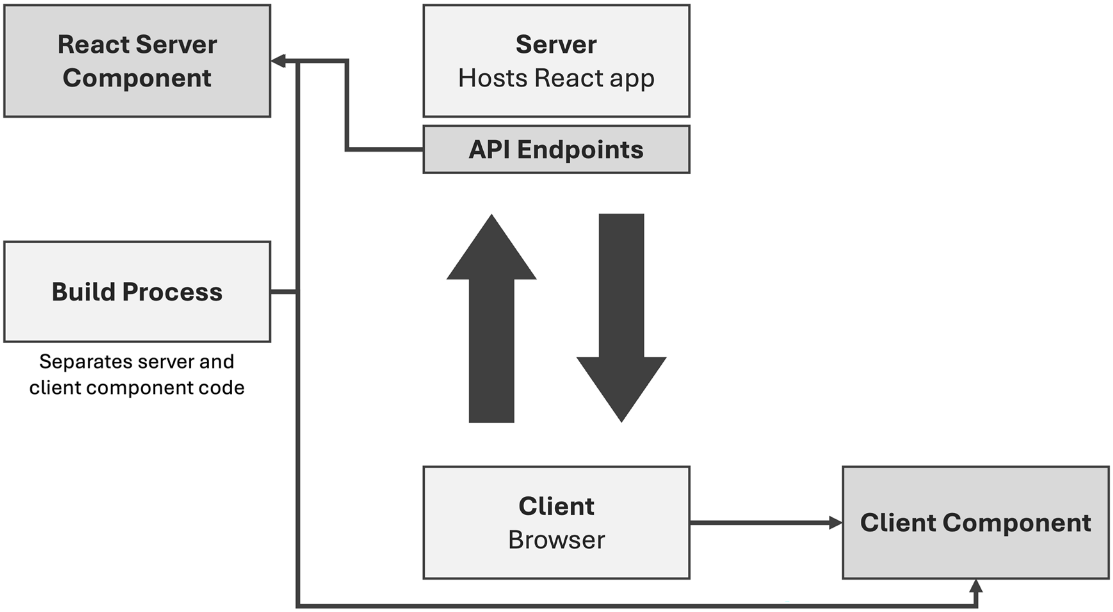
But even though the concepts sound similar, they are quite different.

SSR is all about rendering a component tree to HTML when a request is received. It's about creating an initial page snapshot, in the end.

In addition, when building an interactive web application, a vital part of SSR is that the pre-rendered HTML snapshot gets hydrated on the client side—as explained in the previous chapter (see the *Making Sense of Server-side Rendering (SSR)* section and *Figure 15.3*).

As a result, when using SSR, the entire component tree with all its component functions is evaluated on the server side as well as on the client side. There is no split between server-side and client-side code—it's the same app and the same component tree on both sides. For that reason, you also can't have any server-exclusive code in your React components.

With RSCs, that changes. The code of their component functions, as explained in the previous sections, never ends up on the client side.



That's why an SSR-enabled project doesn't automatically support RSCs. On the other hand, you could set up a project that supports RSCs but also uses SSR for some components—components that should be pre-rendered on the server but that are also needed on the client side (e.g., because they add interactivity to the page). These types of components will be explored in the next section.

It's also worth noting that RSCs, like server-side rendered components in SSR projects, only execute once per request. However, RSCs, unlike "normal" components rendered via SSR, can be executed on-demand while the app is running. They're not limited to being called to create an initial page snapshot.

There is an important question, though: how can you add interactivity, and, for example, handle user input, in React apps where all components are rendered on the server? User interaction takes place in the browser, after all.

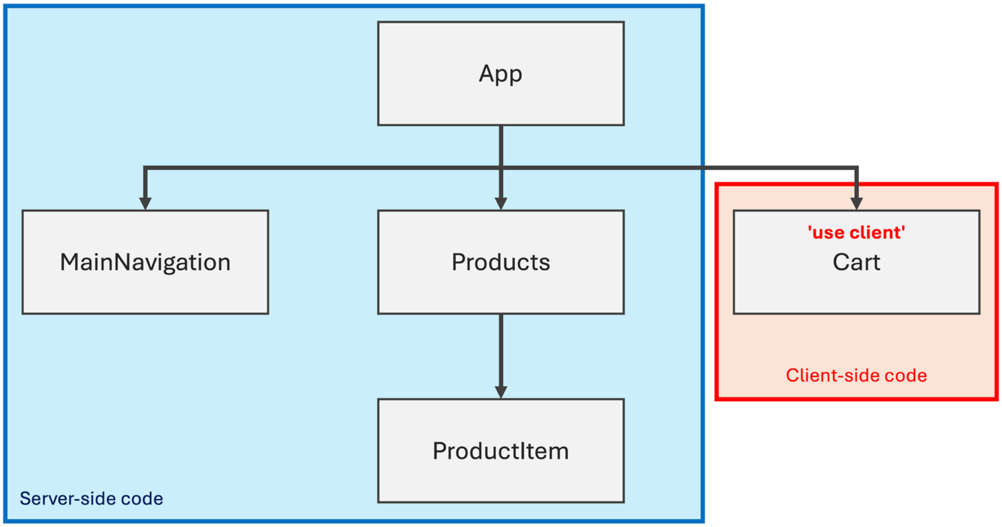
RSCs vs Client Components

RSCs provide some convincing advantages (see the *Making Sense of RSCs* section), but they also introduce one potentially big problem: if all the component code "lives" and executes on the server, there's no room for client-side interactivity.

Not All Components Should Be RSCs

If you have a component that needs to manage some state (e.g., some shopping cart that should only be shown upon user interaction), that state and the UI must be managed and updated by client-side React. Because that was (and is) one of the main selling points of React: you can use it to build highly reactive and interactive UIs. But this goal clearly clashes with the idea of RSCs, where no component code makes it to the browser, and where components are only rendered once per request.

That's why React allows you to define so-called **server-client boundaries** by adding the `'use client'` directive at the top of files that contain component functions that should run on the client side.



You already encountered `'use client'` in the previous chapter, in the *Highlighting Active Links & Using the 'use client' Directive* section. Back then, this directive didn't make a lot of sense. Now, with your newly gained knowledge about RSCs, the purpose behind this directive will become clearer.

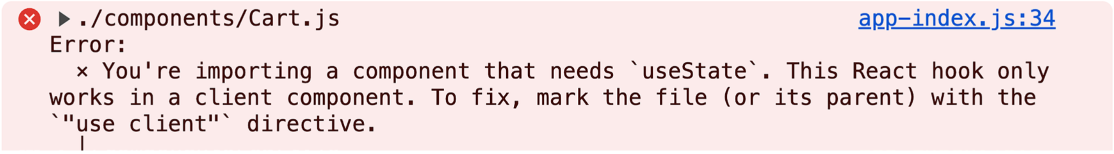
With `'use client'` added to a component file, the components defined in that file become client components. Client components are also pre-rendered on the server, but their code executes on the client side, too. They are hydrated, as explained in the previous chapter. Thus, unlike the code of server components, the code of client components makes it to the client side:

```
'use client';

import { useState } from 'react';

export default function Cart() {
  const [isVisible, setIsVisible] = useState(false);

  function handleCartVisibility() {
    setIsVisible((prevState) => !prevState);
  }

  return (
    <div id="cart">
      <button onClick={handleCartVisibility}>
        {isVisible ? 'Hide Cart' : 'Show Cart'}
      </button>
      {isVisible && <p>Cart Items</p>}
    </div>
  );
}
```

In this example, the `Cart` component is a client component because `'use client'` is added at the top of the file. This is required because the `Cart` component uses the `useState()` Hook, which only works in the browser.

Whenever you add the `'use client'` directive to a component file, the component functions in that file will be included in the client-side code bundle. Thus, the component functions can (and will) be executed in the browser—therefore you can use features that rely on running there, like `useState()` or code that should run upon user input (e.g., if a `<button>` was pressed).

That's also why Next.js shows an error if you try to use a Hook in a component that's not marked as a client component via `'use client'`.



```
▶ ./components/Cart.js                                     app-index.js:34
Error:
  × You're importing a component that needs `useState`. This React hook only
works in a client component. To fix, mark the file (or its parent) with the
`"use client"` directive.
```

Figure 16.8: Next.js complains about the usage of the useState() Hook in an RSC

This error occurs because you're trying to build something impossible: a component that's only evaluated on the server but that also reacts to user input and updates some state. Since the latter, as you learned in *Chapter 4*, *Working with Events and State*, will typically result in a UI update, the code needs to execute on the client side—something that's clearly in conflict with the goal of running the component code only on the server.

Thus, `'use client'` must be added whenever you have a component that needs to run in the browser.

> **Note**
>
> Of course, you will not need to add the `'use client'` directive in projects that don't implement RSCs. That's why you didn't see it in any other React project in earlier chapters.

'use client' Affects Child Components, Too!

Using the `'use client'` directive in a component file has one very important implication: all nested components become client components, too—even if you don't use `'use client'` in their component files.

This is technically necessary since the JSX code of client components is re-evaluated, and all custom components used there are re-executed, every time the client component function is called again (e.g., because of some state change)—that's something you learned in *Chapter 10*, *Behind the Scenes of React and Optimization Opportunities*.

As a result, all the components nested inside a client component must be client components themselves since their code would otherwise not be available on the client side.

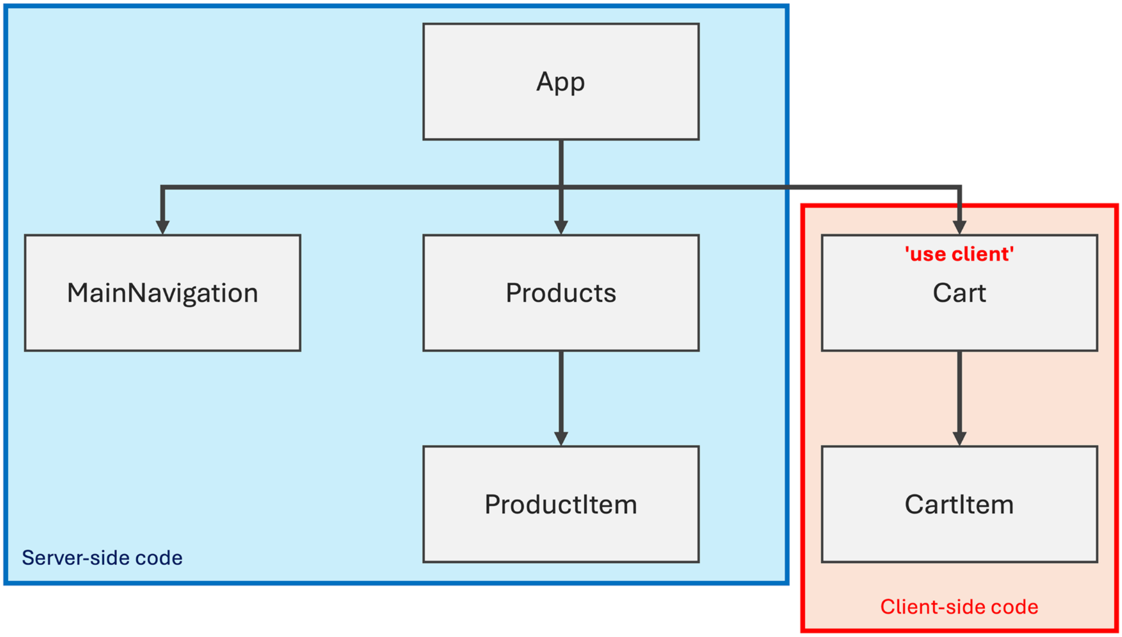

App
MainNavigation
Products
ProductItem
Server-side code
'use client'
Cart
CartItem
Client-side code

Figure 16.9: Child components of client components become client components, too

To keep the client code bundle small and performant, it's typically a good idea to maximize the number of server components and thus minimize the number of client components. Since nested components of client components become client components automatically, you should therefore try to move the server-client boundary (i.e., the usage of `'use client'`) as far down the component tree as possible. Ideally, only the leaves of your component tree use React Hooks or handle user input. Put in other words: only use `'use client'` when you must and try to affect as few components with it as possible.

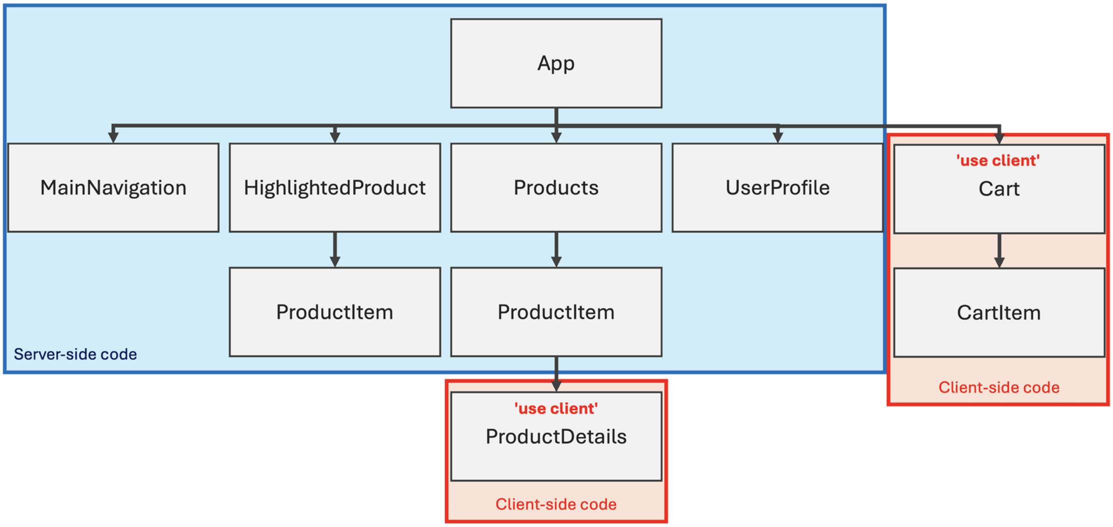

App
MainNavigation
HighlightedProduct
Products
UserProfile
'use client'
Cart
ProductItem
ProductItem
CartItem
Server-side code
Client-side code
'use client'
ProductDetails
Client-side code

Figure 16.10: The majority of components are RSCs

Figure 16.10 shows an example component tree where only a small subset of all components are client components.

The question therefore is this: how can you combine and optimize the usage of server and client components in React projects that support RSCs?

Combining RSCs and Client Components

Typically, you'll end up with React projects where most components don't need to be client components (therefore, they should be RSCs), but where some component functions do need to run in the browser (i.e., they do need `'use client'`).

You can think of `'use client'` marking the point in the component tree where the component type switches from server to client component (see *Figure 16.9* and *Figure 16.10*).

For that reason, React allows you to combine both kinds of components in the same project, though, you need to follow a couple of important rules:

- Server components may import and render client components (i.e., output a client component in their JSX code).
- Client components must not directly import and render server components that rely on server-side features.
- Client components may implicitly render server components via props (e.g., via the `children` prop).

To make these rules a bit less abstract, each case will be shown with a concrete example.

Outputting Client Components in Server Components

You can use client components in the JSX code of server components without issues.

Consider the following example `UserTodos` component, which allows users to manage an array of to-dos that's stored locally via `localStorage`:

```
'use client';

import { useEffect, useRef, useState } from 'react';

export default function UserTodos() {
  const todoRef = useRef(null);
  const [todos, setTodos] = useState([]);

  useEffect(() => {
    const storedTodos = localStorage.getItem('todos');
    setTodos(storedTodos ? JSON.parse(storedTodos) : []);
  }, []);

  function handleAddTodo(event) {
    event.preventDefault();
    const todo = todoRef.current.value.trim();

    const newTodo = {
      id: new Date().getTime(),
      text: todo,
    };

    setTodos((prevTodos) => [...prevTodos, newTodo]);
    const storedTodos = localStorage.getItem('todos');
    localStorage.setItem(
      'todos',
      JSON.stringify(
        storedTodos
          ? [...JSON.parse(storedTodos), newTodo]
          : [newTodo]
      )
    );
  }

  return (
    <>
```

```
      <form onSubmit={handleAddTodo}>
        <input type="text" placeholder="Your to-do" ref={todoRef} />
        <button type="submit">Add</button>
      </form>
      <ul>
        {todos.map((todo) => (
          <li key={todo.id}>{todo.text}</li>
        ))}
      </ul>
    </>
  );
}
```

Since `localStorage` (a browser API), refs, state (`todos` via `useState()`), and event listeners (`submit` via `onSubmit`) are used, this must be a client component. That's why `'use client'` is added at the top of the file.

However, this component can be used in a server component without issues:

```
import UserTodos from '../components/UserTodos';

export default function Home() {
  return (
    <main>
      <h1>Manage your to-dos with ease!</h1>
      <UserTodos />
    </main>
  );
}
```

That's possible because client components can also be rendered on the server—they're just not exclusive to that environment (unlike RSCs, which are). Put in other words: client components are rendered on the server like all components were in SSR projects that do not support RSCs (e.g., the Vite-based SSR-enabled project from the previous section). An initial snapshot is rendered upon the first request, thereafter client-side React takes over and hydrates the component.

Note

In the preceding example, data is loaded from `localStorage` via `useEffect()`. This is done to ensure that the code runs on the server. Since `localStorage` is not available there, accessing it without wrapping with `useEffect()` would cause an error.

Since `useEffect()` is ignored on the server, it's a safe way of using browser-exclusive APIs.

Outputting Server Components in Client Components

As already mentioned in the *'use client' Affects Child Components, Too!* section, you can't import server components into client components and render them there.

Though, in many situations, you'll not get an error. For example, you might have a client-side `Cart` component defined like this:

```
'use client';

import { useState } from 'react';

import CartItem from './CartItem';

export default function Cart() {
  const [isVisible, setIsVisible] = useState(false);

  function handleCartVisibility() {
    setIsVisible((prevState) => !prevState);
  }

  return (
    <div id="cart">
      <button onClick={handleCartVisibility}>
        {isVisible ? 'Hide Cart' : 'Show Cart'}
      </button>
      {isVisible && (
        <ul>
          <CartItem title={'Book'} quantity={1} />
          <CartItem title={'Pen'} quantity={2} />
          <CartItem title={'Pencil'} quantity={5} />
        </ul>
      )}
    </div>
  );
}
```

Unlike `Cart`, the `CartItem` component function might be a server component (i.e., it's not marked via `'use client'`):

```
export default function CartItem({ title, quantity }) {
  return (
    <li>
      <article>
```

```
        <h2>{title}</h2>
        <p>Quantity: {quantity}</p>
      </article>
    </li>
  );
}
```

This code works because the component that used to be a server component (`CartItem`) simply becomes a client component once it is imported and used in a client component file.

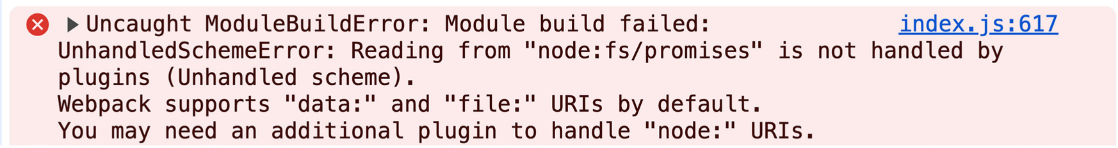
You will, however, face an error message if you're trying to import and use a server component that uses server component-specific features, like a Node.js API or `async/await`.

For example, the following adjusted `DynamicCartItem` component tries to use Node's `fs` package to load a cart item from a file:

```
import fs from 'node:fs/promises';

export default async function DyncamicCartItem({ id }) {
  const data = await fs.readFile(`data/cart.json`, 'utf8');
  const storedCart = JSON.parse(data);
  const cartItem = storedCart.find((item) => item.id === id);

  return (
    <li>
      <article>
        <h2>{cartItem.title}</h2>
        <p>Quantity: {cartItem.quantity}</p>
      </article>
    </li>
  );
}
```

Importing and using this component in the `Cart` component will cause an error.

Trying to run this code will lead to an error message being shown on the screen because React fails to automatically convert `CartItem` to a client component (due to the usage of RSC-exclusive features). Therefore, it'll complain about some server-side code (e.g., some Node.js API) you're trying to use on the client side.



```
▶ Uncaught ModuleBuildError: Module build failed:            index.js:617
UnhandledSchemeError: Reading from "node:fs/promises" is not handled by
plugins (Unhandled scheme).
Webpack supports "data:" and "file:" URIs by default.
You may need an additional plugin to handle "node:" URIs.
```

Figure 16.11: React complains about the usage of a Node.js API in the browser

Hence, in situations like this, you'll need to restructure your application to end up with a valid component combination again. For example, by passing server components as props to client components, instead of directly importing and rendering them.

Rendering Server Components via Props

You can't import and use server components that perform some server-side exclusive operation (like using Node.js APIs) in client components.

But you can change your client component code to not directly import and use the server component. Instead, you can expect to get a server component as a prop—for example, via the special `children` prop about which you learned in *Chapter 3, Components and Props*:

```
'use client';

import { useState } from 'react';

export default function Cart({ children }) {
  const [isVisible, setIsVisible] = useState(false);

  function handleCartVisibility() {
    setIsVisible((prevState) => !prevState);
  }

  return (
    <div id="cart">
      <button onClick={handleCartVisibility}>
        {isVisible ? 'Hide Cart' : 'Show Cart'}
      </button>
      {isVisible && <ul>{children}</ul>}
    </div>
  );
}
```

This adjusted `Cart` component is still a client component. However, since it no longer directly imports and renders the `DynamicCartItem` server component, React is happy.

Instead, the `DynamicCartItem` component is now imported and output in the `Home` component like this:

```
import DyncamicCartItem from '../components/DynamicCartItem';
import Cart from '../components/Cart';

export default function Home() {
  return (
    <>
      <header>
        <Cart>
          <DyncamicCartItem id={1} />
          <DyncamicCartItem id={2} />
          <DyncamicCartItem id={3} />
        </Cart>
      </header>
      <main>
        <h1>Some dummy app</h1>
      </main>
    </>
  );
}
```

The `DynamicCartItem` elements are passed as a value for the `children` prop to the `Cart` component.

This might be unintuitive at first but it's vital to understand that this works because the `DynamicCartItem` components are now rendered as part of another server component—the `Home` component. It's the result of that rendering process that's then passed as a value to the `Cart` component. That component therefore does not include the `DynamicCartItem` component in its part of the component tree. Instead, both `Cart` and `DynamicCartItem` are direct children of the `Home` component.

The overall application component tree would look like this:

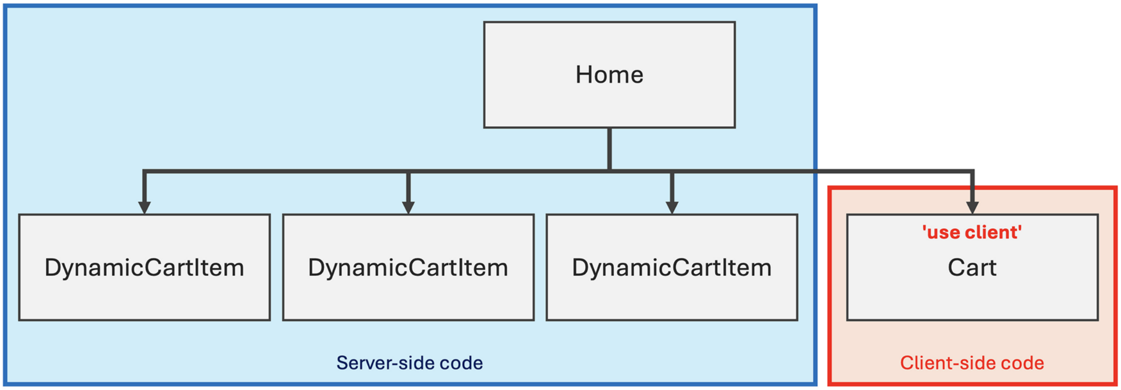


Figure 16.12: DynamicCartItem and Cart are both direct child components of the Home component

Even though, in the finished UI, it might look as if the `DynamicCartItem` is a child of `Cart`, technically, it's not.

It's key to understand that wrapping a component with another component (`<Cart><DynamicCartItem /></Cart>`) leads to a different component tree structure than rendering a component inside another component.

This is therefore a pattern that can be useful in situations where you might need to include a server component in a client component.

Overall, you are able to combine RSCs and client components as needed. Furthermore, Next.js also provides some additional features that can help with RSCs and data fetching via RSCs.

Advanced Data Fetching with Next.js

As mentioned before, in the *Making Sense of RSCs* section, data fetching via RSCs offers various advantages compared to data fetching in client components. You don't have to use `useEffect()` to send HTTP requests to separate backend APIs, you can directly reach out to a database, you can use `async/await`, and so on. Therefore, it's absolutely recommended to fetch data via RSCs whenever possible.

When working with Next.js, RSC-based data fetching becomes even easier because Next.js helps with showing fallback content while you're waiting for data to arrive.

Managing Loading States with Next.js

When working with Next.js (with the App Router), you can define `loading.js` files inside the `app/` folder to set up components that will be rendered while sibling or nested server components are loading data. Next.js determines whether a component is loading data or not by checking whether it returns a `Promise` that hasn't resolved yet.

> **Note**
>
> The next chapter will dive even deeper into handling loading states and showing fallback content. It will explore React's Suspense feature, which allows for granular loading state management as data streams in.

Consider this example `GoalsPage` component, which fetches data from a file:

```
import fs from 'node:fs/promises';

async function fetchGoals() {
  await new Promise((resolve) => setTimeout(resolve, 3000)); // delay

  const goals = await fs.readFile('./data/user-goals.json', 'utf-8');
  return JSON.parse(goals);
}

export default async function GoalsPage() {
  const fetchedGoals = await fetchGoals();

  return (
    <>
    <h1>Top User Goals</h1>
    <ul>
      {fetchedGoals.map((goal) => (
        <li key={goal}>{goal}</li>
      ))}
    </ul>
    </>
  );
}
```

The function (`fetchGoals()`) that performs the actual data fetching has a delay built-in to simulate a slow database or network connection.

Without a `loading.js` file added to the project, the user will stare at a blank or outdated page for a couple of seconds before the requested page is rendered.

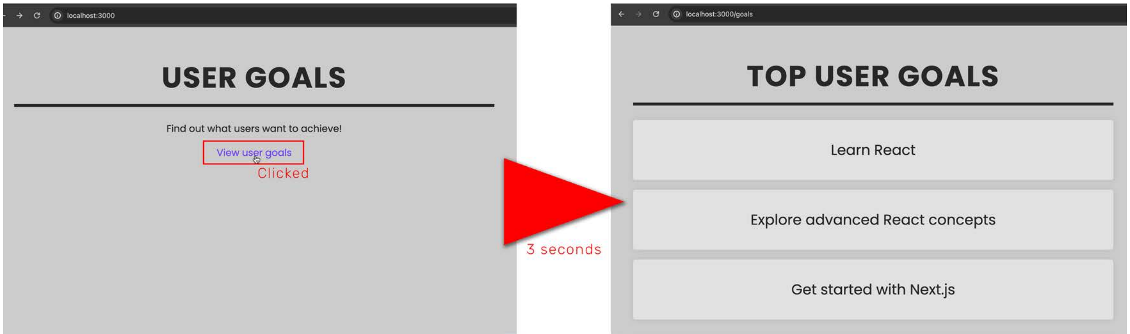

localhost:3000
USER GOALS
Find out what users want to achieve!
View user goals
Clicked
3 seconds
localhost:3000/goals
TOP USER GOALS
Learn React
Explore advanced React concepts
Get started with Next.js

Figure 16.13: After clicking the link, it takes three seconds for the new page to load

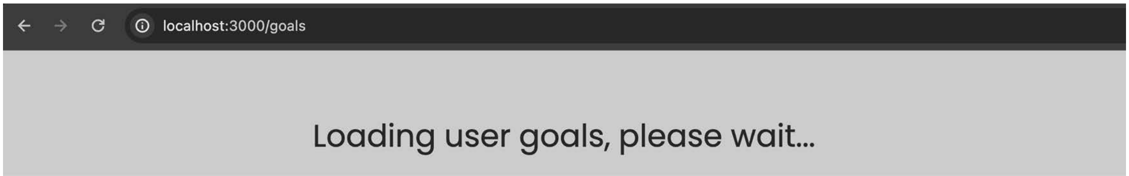
This behavior occurs because the new page is not ready yet and can't be rendered since it's still fetching data.

To improve the user experience, a `loading.js` file can be added next to the slow `app/goals/page.js` file (or, if necessary, in some parent folder, since `loading.js` will also display its content for child routes).

Inside the newly created `app/goals/loading.js` file, a regular React component is created. Like all components in Next.js projects, this is an RSC by default:

```
export default function LoadingGoals() {
  return <p id="fallback">Loading user goals, please wait...</p>;
}
```

The component name (`LoadingGoals`) does not matter. But this component now ensures that the `Loading user goals, please wait…` fallback text is shown on the screen while the user waits for the `GoalsPage` to load and render.



Loading user goals, please wait...

Figure 16.14: The loading fallback content is shown while the page is transitioning

Of course, you can show any fallback content of your choice—it doesn't have to be some simple text as in this example.

Therefore, when working with Next.js, adding `loading.js` files to define fallback components can tremendously improve the experience of your website users.

Besides fetching data, many React apps also need to change data at some point.

From Data Fetching to Data Mutations

At this point, you have learned a lot about RSCs, client components, and how they can (and cannot) work together. In the *Making Sense of RSCs* section, you also learned about some advantages offered by RSCs.

Of course, you also might want to change data, though—not just load and display it.

Handling Data Mutations with Server Actions

React does not just provide support for RSCs; it also allows you to add so-called **Server Actions** to your applications.

Server Actions build up on the same idea as client (form) actions, which were introduced and explained in *Chapter 9*, *Handling User Input & Forms with Form Actions*. However, Server Actions, as their name implies, will execute on the server side, not on the client side.

Hence, you can use Server Actions to retrieve submitted user input on the server and process it there. For example, you could store the submitted data in a file or database.

Consequently, Server Actions are an important building block when aiming to build fully integrated fullstack React applications. Typically, data fetching alone is not enough, which is why the Server Actions feature exists. By having both, RSCs and Server Actions, you're able to fetch and mutate data on the server, while still enabling interactive client-side user experiences where needed.

Unlocking Server Actions in React Projects

Like RSCs, you can't use Server Actions in all React projects. Instead, a special project setup is required to use this feature. For example, Next.js projects support Server Actions out of the box (when using the App Router). Just as with RSCs, you can think of Next.js doing the following things:

- The build workflow and bundling process separate the code that belongs to Server Actions so that it doesn't end up in the client-side bundle.
- Next.js sets up API endpoints that trigger the Server Action functions and respond with any return values defined in those functions.

- Next.js calls these endpoints when needed (e.g., when submitting a form that's connected to a Server Action—as shown in the next section).

Therefore, Server Actions, like RSCs, can be tricky to support in custom projects that do not use Next.js. It's absolutely possible to create custom projects that provide support for both Server Actions and RSCs, but it's not trivial.

Thankfully, using Server Actions (in projects that support them) is not complicated, though.

Defining and Triggering Server Actions

As mentioned in the *Handling Data Mutations with Server Actions* section, Server Actions are very similar to the client form actions you already know from *Chapter 9*.

But there are two key differences that must be considered when creating a Server Action:

- A Server Action function must be asynchronous— (i.e., it must use `async/await`). There are no synchronous Server Actions.
- Inside the Server Action function, at the very beginning of the function body, you must add the `'use server'` directive.

A valid Server Action can therefore be defined and used in a component like this:

```
export default function UserFeedback() {
  async function saveFeedback(formData) {
    'use server';
    const feedback = formData.get('feedback');
    console.log(feedback);
  }

  return (
    <form action={saveFeedback}>
      <p>
        <label htmlFor="feedback">Your feedback</label>
        <textarea id="feedback" name="feedback" rows={3} />
      </p>
      <p><button>Submit</button></p>
    </form>
  );
}
```

As you can see, besides the fact that it must be asynchronous and that it uses the `'use server'` directive, this action function looks like the ones you saw in *Chapter 9*. It receives a `formData` object that will be provided by React when the form is submitted, and you set the `action` function as a value for the `action` prop on a `<form>` element.

As mentioned in the previous section, if you were to search for this code in the code files downloaded by the browser, you wouldn't find it—this code really only runs on the server side.

Note

The `UserFeedback` component from the previous example is an RSC.

If you think about it, this might be strange, though. After all, this component does handle some user input and interaction. Why does it work without `'use client'` then?

Because Server Actions (bound to the `<form>`'s `action` prop) are special. React explicitly supports this pattern inside of RSCs. `'use client'` is indeed required for any other kind of user input handling (e.g., if you rely on the `onSubmit` or `onChange` props). But binding Server Actions via the `action` prop is supported.

Furthermore, it's important to understand that the `'use server'` directive only exists to mark actions as Server Actions. You, for example, can't use it to mark components as server components.

Of course, the preceding example Server Action currently only logs the input to the console. A more realistic action would probably store that data somewhere and redirect the user to some other page.

Handling User Input & Updating the UI

Consider this updated version of the previous example:

```
import { storeFeedback } from '../lib/feedback-db';

function UserFeedback() {
  async function saveFeedback(formData) {
    'use server';
    const feedback = formData.get('feedback');
    storeFeedback(feedback);
  }

  return (
    <form action={saveFeedback}>
      <p>
        <label htmlFor="feedback">Your feedback</label>
        <textarea id="feedback" name="feedback" rows={3} />
      </p>
      <p><button>Submit</button></p>
    </form>
  );
}
```

The `saveFeedback()` Server Action now stores the extracted feedback via the `storeFeedback()` function.

This function is defined like this:

```
import fs from 'node:fs/promises';

export async function storeFeedback(text) {
  const storedFeedback = await fs.readFile('data/user-feedback.json');
  const feedback = JSON.parse(storedFeedback);

  feedback.push({ id: new Date().getTime(), text });

  await fs.writeFile(
    'data/user-feedback.json',
    JSON.stringify(feedback)
  );
}
```

In a real app, data might be stored in a database. Here, in this simple example, it's simply stored in a `user-feedback.json` file that's part of the overall Next.js project.

As you can tell, just as you can directly reach out to a file or database from inside an RSC, you are able to directly edit a file or send a database query from inside a Server Action.

You can also update the UI by programmatically navigating the user to a different page thereafter. In a Next.js application, you can use the `redirect()` function provided by Next.js to trigger such a navigation action—for example, right after storing the submitted feedback text:

```
import { redirect } from 'next/navigation';
import { storeFeedback } from '../lib/feedback-db';

export default function UserFeedback() {
  async function saveFeedback(formData) {
    'use server';
    const feedback = formData.get('feedback');
    await storeFeedback(feedback);
    redirect('/thanks')
  }

  // same JSX code as before, hence omitted
}
```

This is a very common pattern when building fullstack applications since you often want to navigate your website users to a different page once they have submitted data.

But you can also use a different pattern and update the UI that contains the form, based on the form submission.

Server Actions and useActionState()

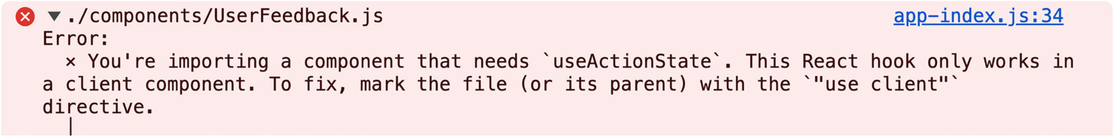
You might remember the `useActionState()` Hook from *Chapter 9*, *Handling User Input & Forms with Form Actions*. This Hook can be used to derive some component state from a (form) action. That state, in turn, can be used to update the UI based on the result of the action.

Since a Server Action is a special kind of form action, you can use that same Hook to update the UI based on your Server Action and its returned values.

For example, you could try using `useActionState()` in the `UserFeedback` component like this:

```
import { useActionState } from 'react';
import { redirect } from 'next/navigation';

import { storeFeedback } from '../lib/feedback-db';
import FeedbackForm from './FeedbackForm';

export default function UserFeedback() {
  async function saveFeedback(prevState, formData) {
    'use server';
    const feedback = formData.get('feedback');

    if (!feedback || feedback.trim() === '') {
      return { error: 'Please provide some feedback!' };
    }

    await storeFeedback(feedback);
    redirect('/thanks');
  }

  const [formState, formAction] = useActionState(saveFeedback, {
    error: null,
  });

  return (
    <form action={formAction}>
      <p>
        <label htmlFor="feedback">Your feedback</label>
        <textarea id="feedback" name="feedback" rows={3} />
      </p>
      {formState.error && <p id="error">{formState.error}</p>}
      <p>
        <button>Submit</button>
      </p>
    </form>
  );
}
```

However, using this code would cause an error:



```
⊗ ▼./components/UserFeedback.js                                    app-index.js:34
  Error:
    × You're importing a component that needs `useActionState`. This React hook only works in
  a client component. To fix, mark the file (or its parent) with the `"use client"`
  directive.
    |
```

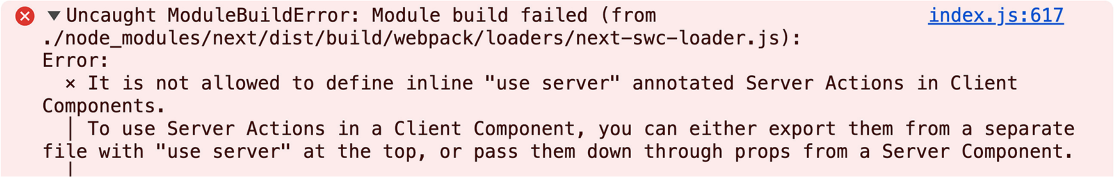
Figure 16.15: React complaints about the usage of a Hook in an RSC

It's an error message you already know from the *Not All Components Should Be RSCs* section and *Figure 16.8*. React does not allow the usage of Hooks in RSCs—and `UserFeedback` is an RSC.

The solution, of course, is straightforward: simply add the `'use client'` directive at the top of the `UserFeedback.js` file:

```
'use client';
import { useActionState } from 'react';
import { redirect } from 'next/navigation';

import { storeFeedback } from '../lib/feedback-db';
import FeedbackForm from './FeedbackForm';

export default function UserFeedback() {
  // component code didn't change, hence omitted
}
```

But with this change applied, you'll encounter another error message:



```
⊗ ▼Uncaught ModuleBuildError: Module build failed (from                  index.js:617
  ./node_modules/next/dist/build/webpack/loaders/next-swc-loader.js):
  Error:
    × It is not allowed to define inline "use server" annotated Server Actions in Client
  Components.
    │ To use Server Actions in a Client Component, you can either export them from a separate
  file with "use server" at the top, or pass them down through props from a Server Component.
    |
```

Figure 16.16: React now complains about the usage of 'use server' and 'use client' in the same file

This error message occurs because the `UserFeedback` component file is currently using both the `'use client'` and `'use server'` directives—in different places, but in the same file.

Put in other words: you can only define a Server Action (and hence use `'use server'`) inside an RSC—not inside a client component.

One possible solution for this problem is to move the feedback form and the `useActionState()` Hook into a new component that will be used as a child component of `UserFeedbackForm`. The Server Action function can then be passed via props to that newly added component.

For example, you can create a `FeedbackForm` component that looks like this:

```
'use client';

import { useActionState } from 'react';

export default function FeedbackForm({action}) {
  const [formState, formAction] = useActionState(action, {
    error: null,
  });

  return (
    <form action={formAction}>
      <p>
        <label htmlFor="feedback">Your feedback</label>
        <textarea id="feedback" name="feedback" rows={3} />
      </p>
      {formState.error && <p id="error">{formState.error}</p>}
      <p>
        <button>Submit</button>
      </p>
    </form>
  );
}
```

This `FeedbackForm` component expects an `action` prop, which is then passed as a value to `useActionState()`. Consequently, the `FeedbackForm` component can be used in the `UserFeedback` component like this:

```
import { redirect } from 'next/navigation';

import { storeFeedback } from '../lib/feedback-db';
import FeedbackForm from './FeedbackForm';

export default function UserFeedback() {
  async function saveFeedback(prevState, formData) {
    'use server';
    const feedback = formData.get('feedback');

    if (!feedback || feedback.trim() === '') {
      return { error: 'Please provide some feedback!' };
    }
    await storeFeedback(feedback);
    redirect('/thanks');
  }

  return <FeedbackForm action={saveFeedback} />;
}
```

If you were to run this code, the application would work without any problems. So, again, just as with RSCs, it's all about coming up with a working component structure.

This is an absolutely valid way of solving this problem. But if you would rather not split the `UserFeedback` component into multiple components and outsource the form into `FeedbackForm`, there is also another possible solution.

Storing Server Actions in Separate Files

You can define Server Actions directly inside of RSCs. As you learned in the previous chapter, you can also pass them around via props.

As an alternative, React also permits storing them in separate files. Doing so allows you to build leaner components since the Server Action code is moved out of the component functions. Furthermore, React is fine with importing a Server Action that's stored in a separate file into a client component file.

Considering the previous code examples, you could move the `saveFeedback()` Server Action into a separate `actions/feedback.js` file in your Next.js project folder—though, the file and folder names are entirely up to you. In that file, you can then also move the `'use server'` directive out of the Server Action and put it at the top of the file:

```
'use server';

import { redirect } from 'next/navigation';

import { storeFeedback } from '../lib/feedback-db';

export async function saveFeedback(prevState, formData) {
  const feedback = formData.get('feedback');

  if (!feedback || feedback.trim() === '') {
    return { error: 'Please provide some feedback!' };
  }

  await storeFeedback(feedback);
  redirect('/thanks');
}
```

Adding the `'use server'` directive at the top of the file enables you to create multiple Server Action functions in that same file. You can then export and use them in any other file they might be needed in.

For example, you can import the `saveFeedback()` action into the `UserFeedback` component, which now doesn't need the separate `FeedbackForm` child component anymore. Since externally stored Server Actions can be imported into client component files without issues, the final `UserFeedback.js` file looks like this:

```
'use client';

import { saveFeedback } from '../actions/feedback';
import { useActionState } from 'react';

export default function UserFeedback() {
  const [formState, formAction] = useActionState(saveFeedback, {
    error: null,
  });
  return (
    <form action={formAction}>
      <p>
        <label htmlFor="feedback">Your feedback</label>
        <textarea id="feedback" name="feedback" rows={3} />
      </p>
      {formState.error && <p id="error">{formState.error}</p>}
      <p>
        <button>Submit</button>
      </p>
    </form>
  );
}
```

Therefore, storing Server Actions in separate files does not just lead to leaner components, but can also help prevent unnecessary component refactoring.

Though, no matter which approach you choose, you can use Server Actions to handle form submissions on the server. Together with RSCs, you can therefore build fullstack applications that seamlessly blend client-side and server-side code.

Summary and Key Takeaways

- React supports two special server-side functionalities: RSCs and Server Actions.
- Both features are not available in React projects unless the project is specifically configured to support them—typically, you'll therefore need to use a framework that supports these features (e.g., Next.js).

- RSCs are components that are never rendered on the client side—instead, they may be rendered on the server (initiated via HTTP requests) or during the build process.
- RSCs return rendering instructions that are picked up by client-side React.
- Since RSCs never run on the client side, you may use server-exclusive APIs and features in them.
- React also permits RSCs to return `Promise` values, hence you can use `async/await` and fetch data asynchronously without issues in RSCs.
- In order to build interactive websites where the UI may change after being rendered, you can also mark components as client components by using the `'use client'` directive.
- Only client components can use Hooks like `useState()` or set up event listeners.
- Client components are also pre-rendered on the server, but unlike RSCs, they may also execute on the client side.
- You may import and use client components inside of RSCs.
- When importing server components into client components, the server components automatically become client components, if possible.
- If an RSC can't be converted into a client component (e.g., because it uses `async/await`), you'll need to restructure the component tree.
- You may pass server components to client components (without converting them) via props.
- React helps with handling form submissions on the server via Server Actions.
- Server Actions work like client actions (see *Chapter 9*) but must be asynchronous (`async/await`) and use the `'use server'` directive.
- You can define Server Actions inside of RSCs or in separate files—in the latter scenario, you can move the `'use server'` directive to the top of the file to define multiple Server Actions in the same file.

What's Next?

In this chapter, you learned about RSCs and Server Actions. You learned that creating and using them is relatively straightforward, but that supporting them in projects is not—hence frameworks like Next.js are commonly used to take advantage of these features.

This chapter gave you an idea of how RSCs and Server Actions work behind the scenes, and which advantages are offered by these features. Throughout this chapter, you also learned about client components and how to combine server and client components. Finally, Server Actions were discussed and different ways of defining and using Server Actions were shown.

The next chapter will build up on this chapter and explore how React's **Suspense** functionality may help with showing fallback content while fetched data is being streamed in.

Test Your Knowledge!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to examples that can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/16-rsc-server-actions/exercises/questions-answers.md`:

1. What is the defining characteristic of React Server Components?
2. What are the problems that React Server Components solve?
3. How are React Server Components created and used in Next.js projects?
4. Why can't React Server Components and Server Actions be used in all React projects?
5. What is the key difference between Server-Side Rendering (SSR) and React Server Components (RSCs)?
6. What is the purpose of the `'use client'` directive?
7. How does the `'use client'` directive affect child components?
8. What are the rules for combining server components and client components?
9. How can you handle loading states in Next.js while fetching data with RSCs?
10. What are Server Actions in React, and how do they differ from client actions?
11. How can you trigger a Server Action?
12. How can you update the UI after a Server Action?
13. Can you define Server Actions in separate files?

Apply What You Learned

With all the newly gained knowledge about Next.js, it's time to apply it to a real demo project—a demo application that will be rendered on the server.

In the following section, you'll find an activity that allows you to practice working with Next.js. As always, you will also need to employ some of the concepts covered in earlier chapters.

Activity 16.1: Build a Mini Blog

In this activity, your job is to build a very simple blog website (with Next.js) that allows users to create and view blog posts. Each blog post should consist of a title, date, and body text. A list of blog post titles and dates should be rendered on the starting page (`/`); upon clicking on a post, users should be taken to the details page (`/blog/<some-id>`), which shows the complete blog post data. A `/blog/new` page should display a form that can be used to create a new post.

Posts should be stored in a `posts.json` file (which may simply store an array of post objects). After creating a new post, users should be redirected to that post's detail page. If users leave either the title or the body field (or both) empty, an error message should be displayed below the form.

Note

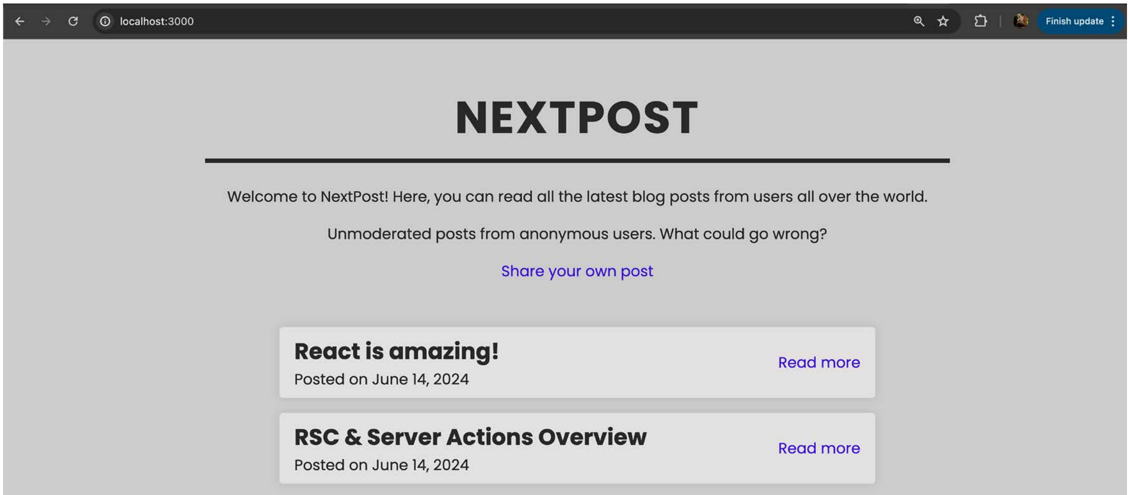
You can find a starting project snapshot for this activity at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/16-rsc-server-actions/activities/practice-1-start`. When downloading this code, you'll always download the entire repository. Make sure to then navigate to the subfolder with the starting code (`activities/practice-1-start`, in this case) to use the right code snapshot.

In the provided starting project, you can explore the `globals.css` file to get an idea of the elements and element structure you might want to use to take advantage of the provided styles. Of course, you can also set up and use your own styles.

After downloading the code and running `npm install` in the project folder to install all required dependencies, the solution steps are as follows:

1. Add three new `page.js` files (and an appropriate folder structure) for the three pages: `/`, `/blog/new`, and `/blog/<some-id>`.
2. Add a new `posts.json` file in a `data/` folder in the root project folder. This file should initially store an empty array.
3. Output a `<form>` with title and body input fields on the `/blog/new` page.
4. Create a new Server Action in a separate file and import and "connect" it to `<form>`. The Server Action should retrieve the entered title and body text, create a new object, which also includes an ID and creation date snapshot, and store that data in the `posts.json` file. Data must be stored such that existing blog posts aren't lost.
5. Update the Server Action to implement input validation and output the validation results above the submit button.
6. Fetch the blog posts on the starting page and output a list of blog posts (title and date). Each post should be clickable and take the user to the details page.
7. On the details page, fetch and output the blog post details (by using the ID).
8. Finally, redirect the user to the appropriate details page from inside the Server Action, after a blog post is created.

The final page should look as shown in the following screenshots:



localhost:3000
Finish update

NEXTPOST

Welcome to NextPost! Here, you can read all the latest blog posts from users all over the world.

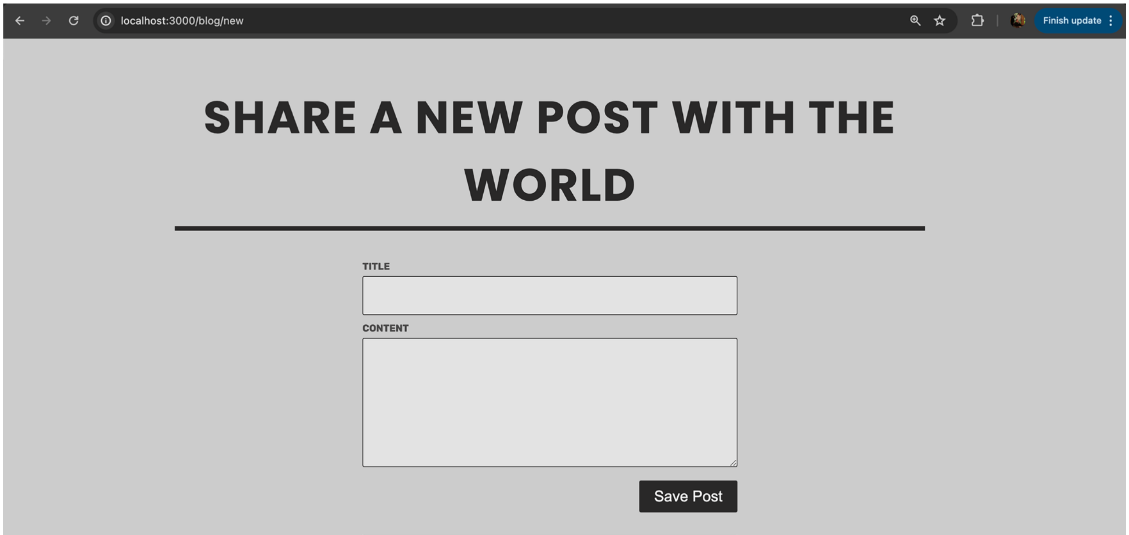
Unmoderated posts from anonymous users. What could go wrong?

Share your own post

React is amazing!
Posted on June 14, 2024
Read more

RSC & Server Actions Overview
Posted on June 14, 2024
Read more

Figure 16.17: The home page, showing a list of blog posts



localhost:3000/blog/new
Finish update

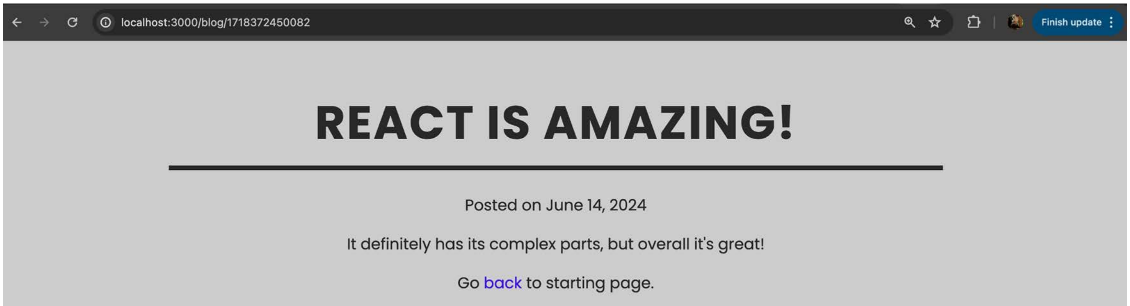
SHARE A NEW POST WITH THE WORLD

TITLE

CONTENT

Save Post

Figure 16.18: The /blog/new page, waiting for user input


localhost:3000/blog/1718372450082
Finish update

REACT IS AMAZING!

Posted on June 14, 2024

It definitely has its complex parts, but overall it's great!

Go back to starting page.

Figure 16.19: The /blog/<some-id> page displaying blog post details

Note

You can find the full code for this activity, and an example solution, here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/16-rsc-server-actions/activities/practice-1`.

17

Understanding React Suspense & The use() Hook

Learning Objectives

By the end of this chapter, you will be able to do the following:

- Describe the purpose and functionality of React's Suspense feature
- Use Suspense with RSCs to show fallback content on a granular level
- Use Suspense for client components via React's `use()` Hook
- Apply different Suspense strategies for data fetching and fallback content

Introduction

In *Chapter 10*, *Behind the Scenes of React and Optimization Opportunities*, in the *Reducing Bundle Sizes via Code Splitting (Lazy Loading)* section, you learned about React's `<Suspense>` component and how it may be used in the context of lazy loading and code splitting to show fallback content while a code bundle is being downloaded.

As explained there, the purpose of the `Suspense` component is to simplify the process of showing fallback content, which, in turn, can lead to a better user experience. Since staring at outdated content or a blank page is not something most users appreciate, having a built-in feature that shows alternative content is very convenient.

In this chapter, you'll learn that React's `Suspense` component is not limited to being used for code splitting. Instead, it can also be used for data fetching to show some temporary content while data is being loaded (e.g., from a database). Though, as you will also learn, `Suspense` can only be used for data fetching if the data is fetched in a certain way.

In addition, this chapter will revisit the `use()` Hook, which was introduced in *Chapter 11*, *Working with Complex State*. As you will learn, besides using it for getting access to context values, this Hook can be used in conjunction with `Suspense` as well.

Showing Granular Fallback Content with Suspense

When fetching data or downloading a resource (e.g., a code file), loading delays can occur—delays that can lead to a bad user experience. You should therefore consider showing some temporary fallback content while waiting for the requested resource.

For that reason, to simplify the process of rendering fallback content while waiting for some resource, React offers its `Suspense` component. As shown in *Chapter 10*, *Behind the Scenes of React and Optimization Opportunities*, you can use the `Suspense` component as a wrapper around React elements that fetch some code or data. For example, when using it in the context of code splitting, you can show some temporary fallback content like this:

```
import { lazy, Suspense, useState } from 'react';

const DateCalculator = lazy(() => import(
    './components/DateCalculator.jsx'
  )
);

function App() {
  const [showDateCalc, setShowDateCalc] = useState(false);

  function handleOpenDateCalc() {
    setShowDateCalc(true);
  }

  return (
    <>
      <p>This app might be doing all kinds of things.</p>
      <p>
        But you can also open a calculator which calculates
        the difference between two dates.
      </p>
      <button onClick={handleOpenDateCalc}>Open Calculator</button>
      <Suspense fallback={<p>Loading...</p>}>
        {showDateCalc && <DateCalculator />}
      </Suspense>
    </>
  );
}
```

In this example (which is from a regular Vite-based React project), React's `Suspense` component is wrapped around the conditionally rendered `DateCalculator` component. `DateCalculator` is created with the help of React's `lazy()` function, which is used to lazily (i.e., on demand) load the code bundle that belongs to this component.

As a result, the entire other page content is shown right from the start. Only the conditionally displayed `DateCalculator` component is replaced with the fallback content (`<p>Loading...</p>`) while the code is being fetched. Thus, `Suspense` is used to render some fallback JSX code on a very granular level. Instead of replacing the entire page or component markup with some temporary content, only a small part of the UI is replaced.

Of course, `Suspense` therefore provides a functionality that would also be nice to have when fetching data—after all, delays occur frequently there, too.

Using Suspense for Data Fetching with Next.js

As explained in the previous chapter, in the *Managing Loading States with Next.js* section, the process of data fetching also often comes with waiting times that can negatively impact user experience. That's why, in that same section, you learned that Next.js allows you to define a `loading.js` file that contains some fallback component that's rendered during such a delay.

However, using that approach essentially replaces the entire page (or the main area of that page) with the loading fallback component content. But that's not always ideal—you instead might want to display some loading fallback content on a more granular level when fetching data.

Thankfully, in Next.js projects, you can use `Suspense` in a similar way, as shown in the example from the previous section, to wrap it around components that fetch data. Since Next.js supports HTTP response streaming, it's able to render the rest of the page immediately while streaming the content that depends on the fetched data to the client side once it's available. Until the data is loaded and available, `Suspense` will render its defined fallback.

Therefore, coming back to the example from the *Managing Loading States with Next.js* section of *Chapter 16, React Server Components & Server Actions*, you can take advantage of `Suspense` by outsourcing the data fetching code into a separate `UserGoals` component:

```
import fs from 'node:fs/promises';

async function fetchGoals() {
  await new Promise((resolve) => setTimeout(resolve, 3000)); // delay

  const goals = await fs.readFile('./data/user-goals.json', 'utf-8');
  return JSON.parse(goals);
}

export default async function UserGoals() {
  const fetchedGoals = await fetchGoals();

  return (
    <ul>
      {fetchedGoals.map((goal) => (
        <li key={goal}>{goal}</li>
      ))}
    </ul>
  );
}
```

This `UserGoals` component can then be wrapped with `Suspense` in the `GoalsPage` component like this:

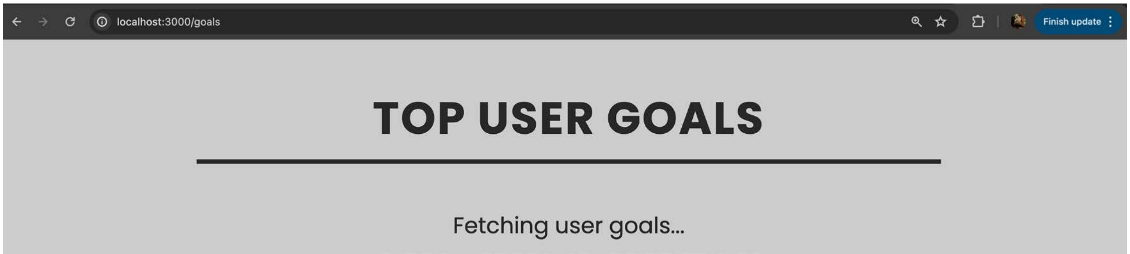
```
import { Suspense } from 'react';

import UserGoals from '../../components/UserGoals';

export default async function GoalsPage() {
  return (
    <>
      <h1>Top User Goals</h1>
      <Suspense fallback={
        <p id="fallback">Fetching user goals...</p>}
      >
        <UserGoals />
      </Suspense>
    </>
  );
}
```

This code now utilizes React's `Suspense` component to show a fallback paragraph while the `UserGoals` component is fetching data.

Note

You can find the complete demo project code on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/17-suspense-use/examples/02-data-fetching-suspense`.

As a result, when users navigate to `/goals`, they immediately see the title (the `<h1>` element) in combination with the fallback content. There is no need for a separate `loading.js` file anymore.



TOP USER GOALS

Fetching user goals...

Figure 17.1: The fallback content is shown as part of the target page, instead of entirely replacing it

However, the advantage of using `Suspense` in this situation is not just that the `loading.js` file isn't needed anymore. Instead, data fetching and fallback content can now be managed on a very granular level.

For example, in a more complex online shop application, you could have a component like this:

```
function ShopOverviewPage() {
  return (
    <>
      <header>
        <h1>Find your next deal!</h1>
        <MainNavigation />
      </header>
      <main>
        <Suspense fallback={<DailyDealSkeleton />}>
          <DailyDeal />
        </Suspense>
        <section id="search">
          <h2>Looking for something specific?</h2>
          <Search />
        </section>
        <Suspense fallback={<p>Fetching products...</p>}>
          <Products />
        </Suspense>
      </main>
    </>
  );
}
```

In this example, the `<header>` and `<section id="search">` elements are always visible and rendered. On the other hand, `<DailyDeal />` and `<Products />` are only rendered once their data has been fetched. Until then, their respective fallbacks are displayed.

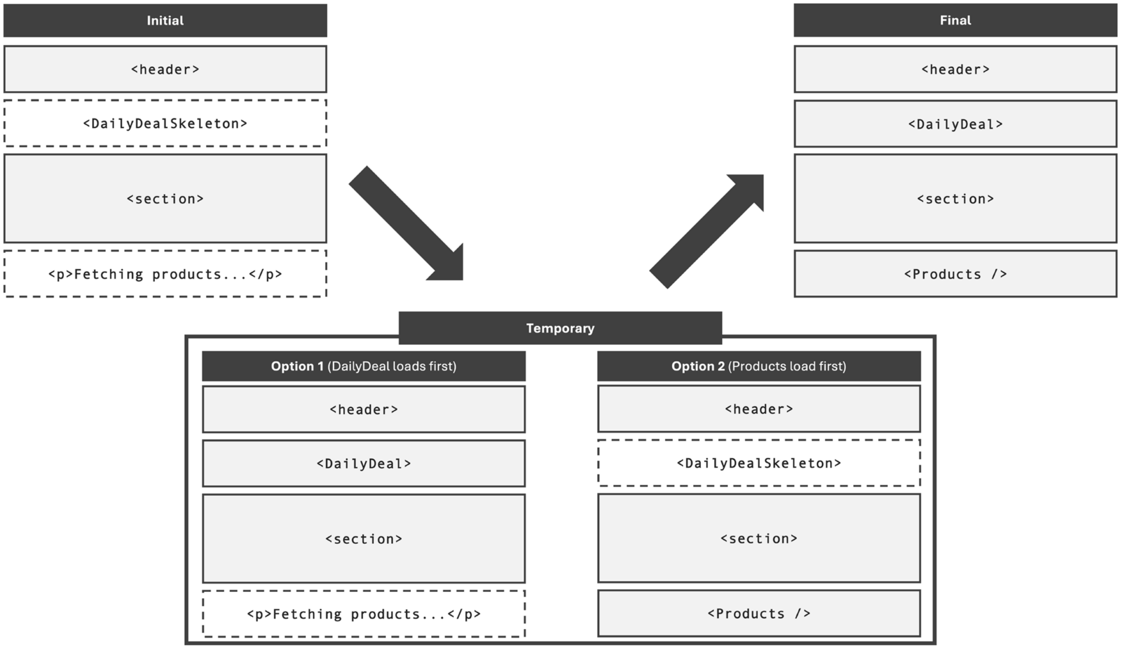


Figure 17.2: Placeholders are shown initially until loaded data is streamed in and rendered to the screen

`<DailyDeal />` and `<Products />` will be loaded and rendered independently from each other since they're wrapped by two different `Suspense` blocks. Consequently, users will immediately see the header and search area, and then eventually see the daily deal and products—though either of the two may load and render first.

What's important about these examples is that the components wrapped by `Suspense` are RSCs that use `async/await`. As you will learn in the next section, not all React components will interact with the `Suspense` component. But React Server Components, in Next.js projects, will.

Using Suspense in Other React Projects—Possible, But Tricky

The previous section explored how you may take advantage of `Suspense` for data fetching with RSCs in Next.js projects.

However, `Suspense` is not a Next.js-specific feature or concept—instead, it's provided by React itself. Consequently, you can use it in any React project to show fallback content while data is being fetched.

At least, that's the theory. But as it turns out, you can't use it with all components and data fetching strategies.

Suspense Does Not Work with useEffect()

Since fetching data via `useEffect()` is a common strategy, you might be inclined to use `Suspense` in conjunction with this Hook to show some fallback content while data is being loaded via the effect function.

For example, the following `BlogPosts` components uses `useEffect()` to load and display some blog posts:

```
import { useEffect, useState } from 'react';

function BlogPosts() {
  const [posts, setPosts] = useState([]);
  useEffect(() => {
    async function fetchBlogPosts() {
      // simulate slow network
      await new Promise((resolve) => setTimeout(resolve, 3000));
      const response = await fetch(
        'https://jsonplaceholder.typicode.com/posts'
      );
      const posts = await response.json();
      setPosts(posts);
    }

    fetchBlogPosts();
  }, []);
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}
```

You could wrap this component with `Suspense` like this:

```
import { Suspense } from 'react';

import BlogPosts from './components/BlogPosts.jsx';

function App() {
  return (
    <>
```

```
      <h1>All posts</h1>
      <Suspense fallback={<p>Fetching blog posts...</p>}>
        <BlogPosts />
      </Suspense>
    </>
  );
}
```

Unfortunately, this will not work in the intended way, though. Instead of displaying the fallback content, nothing will be rendered while the data is being fetched.

The reason for this behavior is that `Suspense` is intended to suspend when fetching data during the component rendering process—not when fetching inside of some effect function.

It helps to recall how `useEffect()` works (from *Chapter 8*, *Handling Side Effects*): the effect function is executed after the component function is executed, i.e., after the first component render cycle is done.

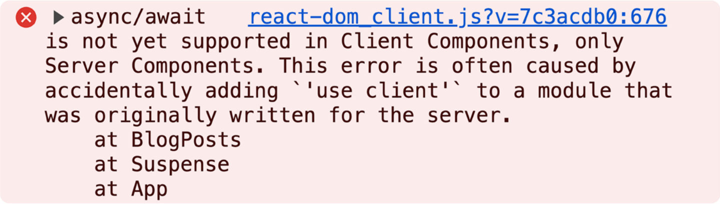
As a result, you can't use `Suspense` to show fallback content when fetching data via `useEffect()`. Instead, in those cases, you need to manually manage and use some loading state in the component that performs the data fetching (i.e., by manually managing different state slices like `isLoading`—for example, as explained and shown in *Chapter 11*, *Working with Complex State*, in the *Limitations of useState()* and *Managing State with useReducer()* sections.

Fetching Data while Rendering—the Incorrect Way

Since `Suspense` intends to show fallback content while a component is fetching data during its rendering process, you could try to re-write the `BlogPosts` component to look like this:

```
async function BlogPosts() {
  await new Promise((resolve) => setTimeout(resolve, 3000));
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts'
  );
  const posts = await response.json();

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}
```

But trying to use this code will yield an error in the browser developer tools:



```
✖ ▸async/await    react-dom_client.js?v=7c3acdb0:676
  is not yet supported in Client Components, only
  Server Components. This error is often caused by
  accidentally adding `'use client'` to a module that
  was originally written for the server.
      at BlogPosts
      at Suspense
      at App
```

Figure 17.3: React complains about async components on the client side

React does not support the usage of `async/await` in client components. Only React Server Components may use that syntax (and therefore return promises). Consequently, regular React projects, which are not set up to support RSCs, can't use this solution.

Of course, you could come up with a (problematic) alternative solution like this:

```
function BlogPosts() {
  const [posts, setPosts] = useState([]);
  new Promise(() => setTimeout(() => {
    return fetch(
      'https://jsonplaceholder.typicode.com/posts'
    ).then(response => response.json())
     .then(fetchedPosts => setPosts(fetchedPosts));
  }, 3000));

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}
```

But this approach was already discarded in *Chapter 8*, *Handling Side Effects*, in the *What's the Problem?* section—the code creates an infinite loop.

So, fetching data as part of a component's rendering process is really difficult when not working with RSCs.

Getting Suspense Support Is Tricky

Since `Suspense` requires data fetching to occur during the rendering process, which is difficult to set up manually, the React documentation (`https://react.dev/reference/react/Suspense#displaying-a-fallback-while-content-is-loading`) itself mentions that *"only Suspense-enabled data sources will activate the Suspense component,"* further stating that those data sources include:

- Data fetching with Suspense-enabled frameworks like Relay and Next.js
- Lazy-loading components code with `lazy()`
- Reading the value of a Promise with `use()`

On the same page, the official documentation highlights that *"Suspense-enabled data fetching without the use of an opinionated framework is not yet supported."*

> **Note**
>
> Documentation may change over time—and so may React. But even though the exact wording may differ at the point of time you're reading this, the way of using `Suspense`, and the fact that it can't be used without special libraries or features like `lazy()`, is highly unlikely to change.
>
> This chapter was written when React 19 was released. You can visit the official changelog of this book to find out whether anything significant has changed since then: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/main/CHANGELOG.md`.

Therefore, unless you plan on building your own Suspense-enabled library, you either have to stick to using `Suspense` for code-splitting (via `lazy()`), use a third-party framework or library that integrates with `Suspense`, or explore the usage of that `use()` Hook.

Of course, the `lazy()` function (and how to use it with `Suspense`) was already covered in *Chapter 10, Behind the Scenes of React and Optimization Opportunities*, in the *Reducing Bundle Sizes via Code Splitting (Lazy Loading)* section. But what about the other two options: `Suspense`-enabled libraries and the `use()` Hook?

Using Suspense for Data Fetching with Supporting Libraries

As you learned in the *Using Suspense for Data Fetching with Next.js* section, you can use `Suspense` for data fetching when working with Next.js. But while Next.js is one of the most popular React frameworks that supports `Suspense`, it's not the only option you have.

For example, TanStack Query (formerly known as React Query) is another popular third-party library that unlocks `Suspense` for data fetching. This library, unlike Next.js, is not a library that aims to help with building full-stack React apps or running code on the server side, though. Instead, TanStack Query is a library that's all about helping with client-side data fetching, data mutations, and asynchronous state management. Since it runs on the client side, it therefore works in React projects that do not integrate with SSR and RSCs, too—although you can also use it in such projects.

TanStack Query is a complex, feature-rich library—we could probably write an entire book about it. But the following short code snippet (which is from a Vite-based project, not from a Next.js project) shows how you may fetch data with the help of that library:

```
import { useSuspenseQuery } from '@tanstack/react-query';

async function fetchPosts() {
  await new Promise((resolve) => setTimeout(resolve, 3000));
  const response = await fetch('https://jsonplaceholder.typicode.com/posts');
  const posts = await response.json();
  return posts;
}

function BlogPosts() {
  const {data} = useSuspenseQuery({
    queryKey: ['posts'],
    queryFn: fetchPosts
  });

  return (
    <ul>
      {data.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}
```

In this example, the `BlogPosts` component uses TanStack Query's `useSuspenseQuery()` Hook, in conjunction with a custom `fetchPosts()` function, to fetch data via an HTTP request. As the name of the Hook implies, it integrates with React's `Suspense` component.

As a result, the `BlogPosts` component can then be wrapped with `Suspense` like this:

```
import { Suspense } from 'react';

import BlogPosts from './components/BlogPosts.jsx';

function App() {
  return (
    <>
      <h1>All posts</h1>
      <Suspense fallback={<p>Fetching blog posts...</p>}>
        <BlogPosts />
      </Suspense>
    </>
  );
}
```

As you can tell, `Suspense` is used in the same way it was used with `lazy()` or Next.js. So, its functionality and usage don't change—if you're wrapping it around a component that integrates with `Suspense` (like `BlogPost` does, via TanStack Query's `useSuspenseQuery()` Hook), `Suspense` can be used to output some fallback content while some data fetching process is underway.

Note

You can find the complete example project on GitHub: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/17-suspense-use/examples/05-tanstack-query`.

Of course, this is just a simple example. You can do more with TanStack Query, and there also are other libraries that can be used in conjunction with `Suspense`. It's just important to understand that there are other options than Next.js. But it's also crucial to keep in mind that not all code (and also not all libraries) will work with `Suspense`.

Besides using libraries that directly integrate with `Suspense` (like TanStack Query via its `useSuspenseQuery()` Hook), you can also use `Suspense` for data fetching with the help of React's built-in `use()` Hook.

use()ing Data while Rendering

The `use()` Hook offered by React is not limited to accessing context values, as shown in *Chapter 11*, *Working with Complex State*—instead, it may also be used to read values from a promise.

Thus, you can use the `use()` Hook during a component's rendering process to extract and use the value of a promise. `use()` will automatically interact with any wrapping `Suspense` component and let it know about the current status of the data fetching process (i.e., if the promise has been resolved or not).

The example from the *Fetching Data while Rendering—the Incorrect Way* section can therefore be adjusted to use the `use()` Hook like this:

```
import { use } from 'react';

async function fetchPosts() {
  await new Promise((resolve) => setTimeout(resolve, 3000));
  const response = await fetch(
    'https://jsonplaceholder.typicode.com/posts'
  );
  const posts = await response.json();
  return posts;
}
```

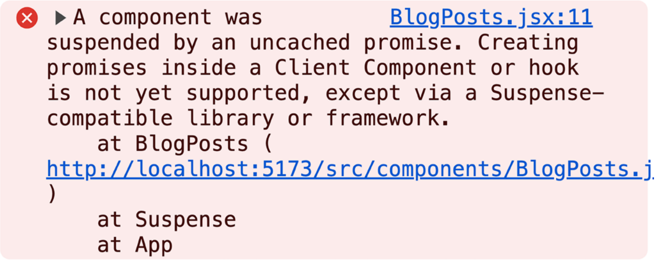
```
function BlogPosts() {
  const posts = use(fetchPosts());

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}
```

The `BlogPosts` component is now no longer a component that uses `async/await`. Instead, it uses the imported `use()` Hook to read the value of the promise produced by calling `fetchPosts()`.

As mentioned, `use()` interacts with `Suspense`, hence `BlogPosts` can be wrapped with `Suspense` like this:

```
import { Suspense } from 'react';

import BlogPosts from './components/BlogPosts.jsx';

function App() {
  return (
    <>
      <h1>All posts</h1>
      <Suspense fallback={<p>Fetching blog posts...</p>}>
        <BlogPosts />
      </Suspense>
    </>
  );
}
```

When running this code, it might work as intended (depending on the React version you're using), but it's more likely to not yield any results or even show an error message in the browser developer tools:



```
▶A component was          BlogPosts.jsx:11
suspended by an uncached promise. Creating
promises inside a Client Component or hook
is not yet supported, except via a Suspense-
compatible library or framework.
    at BlogPosts (
http://localhost:5173/src/components/BlogPosts.j
)
    at Suspense
    at App
```

As explained by this error message, the `use()` Hook is not intended to be used with regular promises as created in the previous example. Instead, it should be used on promises that are provided by *Suspense-compatible* libraries or frameworks.

> **Note**
>
> If you want to go against the official recommendation and try to build promises that support `use()` and `Suspense`, you can explore the official `Suspense` demo projects linked in the official React documentation (`https://19.react.dev/reference/react/Suspense`)—for example, this project: `https://codesandbox.io/p/sandbox/strange-black-6j7nnj`.
>
> Please note that, as mentioned in the documentation, the approach used in that demo project uses unstable APIs and may not work with future React versions.

So, again, support from a third-party framework or library is needed. No matter if you try to use `Suspense` with components that fetch data as part of the rendering process with or without `use()`, you end up needing help.

Put in other words: to take advantage of `Suspense`, you either need to directly fetch data via a Suspense-compatible library or framework, or you need to use the `use()` Hook on a promise that's generated by a Suspense-compatible library or framework.

One such framework is, again, Next.js. Besides using `Suspense` around RSCs, as shown in the section *Using suspense for Data Fetching with Next.js*, you can also use `Suspense` in conjunction with the `use()` Hook on promises produced by Next.js.

Using use() with Promises Created by Next.js

Next.js projects are able to create promises that will work with `use()` and `Suspense`. To be precise, any promise you create in an RSC and pass to a (client) component via props qualifies as a `use()`able promise.

Consider this example code:

```
import fs from 'node:fs/promises';

import UserGoals from '../../components/UserGoals';

async function fetchGoals() {
  await new Promise((resolve) => setTimeout(resolve, 3000)); // delay

  const goals = await fs.readFile('./data/user-goals.json', 'utf-8');
  return JSON.parse(goals);
}

export default function GoalsPage() {
  const fetchGoalsPromise = fetchGoals();
  return (
    <>
      <h1>Top User Goals</h1>
      <UserGoals promise={fetchGoalsPromise} />
    </>
  );
}
```

In this code snippet, a promise is created by calling `fetchGoals()` and stored in a constant called `fetchGoalsPromise`. The created promise (`fetchGoalsPromise`) is then passed as a value for the `promise` prop to the `UserGoals` component.

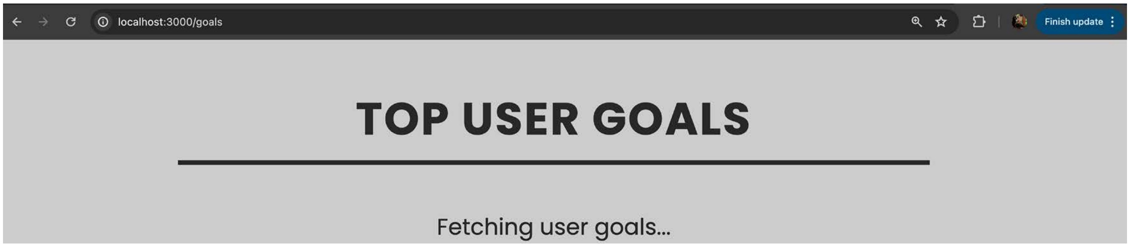
Along with another component, this `UserGoals` component is defined in the `UserGoals.js` file like this:

```
import { use, Suspense } from 'react';

function Goals({ fetchGoalsPromise }) {
  const goals = use(fetchGoalsPromise);

  return (
    <ul>
      {goals.map((goal) => (
        <li key={goal}>{goal}</li>
      ))}
    </ul>
  );
}

export default function UserGoals({ promise }) {
  return (
    <Suspense fallback={<p id="fallback">Fetching user goals...</p>}>
      <Goals fetchGoalsPromise={promise} />
    </Suspense>
  );
}
```

In this code example, the `UserGoals` component uses `Suspense` to wrap the `Goals` component to which it essentially forwards the received `promise` prop value (via the `fetchGoalsPromise` prop). The `Goals` component then reads that promise value via the `use()` Hook.

Since the promise is created in an RSC (`GoalsPage`) that's managed by Next.js, React will not complain about this code—Next.js creates promises that work with `use()`. Instead, it will show the fallback content (`<p id="fallback">Fetching user goals...</p>`) while data is being fetched and renders the final user interface once the data has arrived and has been streamed to the client.

As explained before, any elements not wrapped by Suspense (i.e., the `<h1>` element, in this example) will be displayed right from the start.



TOP USER GOALS

Fetching user goals...

Figure 17.5: The fallback text is shown next to the title while data is fetched via use()

It's also worth noting that both `UserGoals` and `Goals` are RSCs, too—nonetheless, they can use the `use()` Hook.

Normally, Hooks can't be used in RSCs but the `use()` Hook is special. Just as it may be used inside `if` statements or loops (as explained in *Chapter 11*, *Working with Complex State*), it can be executed in both server and client components.

However, when working with a server component, you can also simply use `async/await` instead of `use()`. Thus, the `use()` Hook is really only useful when it comes to reading promise values in client components—there, `async/await` is not available.

Using use() in Client Components

Besides using it for accessing context, the `use()` Hook was introduced to help with reading values from promises in client components—i.e., in situations where you can't use `async/await`.

Consider this updated *user goals* example, where some state is managed and a side effect is triggered:

```
'use client';

import { use, Suspense, useEffect, useState } from 'react';

// sendAnalytics() is a dummy function that just logs to the console
import { sendAnalytics } from '../lib/analytics';

function Goals({ fetchGoalsPromise }) {
  const [mainGoal, setMainGoal] = useState();
  const goals = use(fetchGoalsPromise);

  function handleSetMainGoal(goal) {
    setMainGoal(goal);
  }

  return (
    <ul>
      {goals.map((goal) => (
        <li
          key={goal}
          id={goal === mainGoal ? 'main-goal' : undefined}
          onClick={() => handleSetMainGoal(goal)}
        >
          {goal}
        </li>
      ))}
    </ul>
  );
}

export default function UserGoals({ promise }) {
  useEffect(() => {
    sendAnalytics('user-goals-loaded', navigator.userAgent);
  }, []);

  return (
    <Suspense fallback={<p id="fallback">Fetching user goals...</p>}>
      <Goals fetchGoalsPromise={promise} />
    </Suspense>
  );
}
```

In this example, the `Goals` component uses `useState()` to manage the information of which goal was marked as the main goal by the user. Furthermore, the `UserGoals` component (which uses `Suspense`) utilizes the `useEffect()` Hook to send an analytics event once the component renders (i.e., before the suspended `Goals` component is displayed). Due to the usage of all these client-side exclusive features, the `'use client'` directive is required.

As a result, `async/await` can't be used in the `Goals` and `UserGoals` components. But since the `use()` Hook can be used in client components, it offers a possible solution for situations like this. And, since this example is from a Next.js application, React will not complain about the kind of promise being consumed by `use()`. Instead, this example code would lead to the fallback content being displayed while the goals data is fetched.

Suspense Usage Patterns

As you have learned, the `Suspense` component can be wrapped around components that fetch data as part of their rendering process—as long as they do it in a compliant way.

Of course, in many projects, you may have multiple components that fetch data and that should display some fallback content while doing so. Thankfully, you can use the `Suspense` component as often as needed—you can even combine multiple `Suspense` components with each other.

Revealing Content Together

Thus far, in all examples, `Suspense` was always wrapped around exactly one component. But there is no rule that would stop you from wrapping `Suspense` around multiple components.

For example, the following code is valid:

```
function Shop() {
  return (
    <>
      <h1>Welcome to our shop!</h1>
      <Suspense fallback={<p>Fetching shop data...</p>}>
        <DailyDeal />
        <Products />
      </Suspense>
    </>
  );
}
```

In this code snippet, data fetching in the `DailyDeal` and `Products` components starts simultaneously. Since both components are wrapped by one single `Suspense` component, the fallback content is displayed unti both components are done fetching data. So, if one component (e.g., `DailyDeal`) is done after one second, and the other component (`Products`) takes five seconds, both components are only revealed (and replace the fallback content) after five seconds.

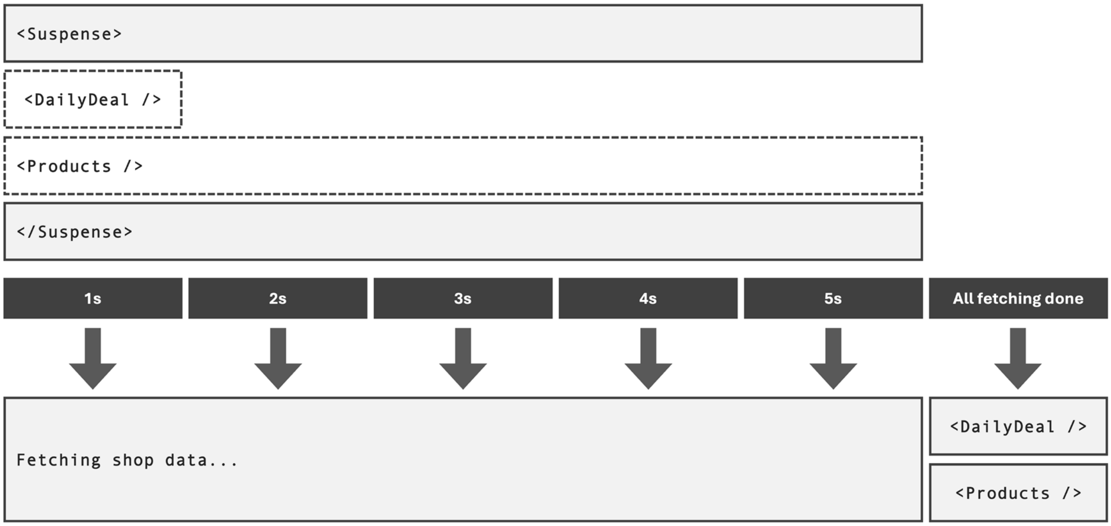


Figure 17.6: Data is fetched in parallel, and fallback content is shown via Suspense until all components are done

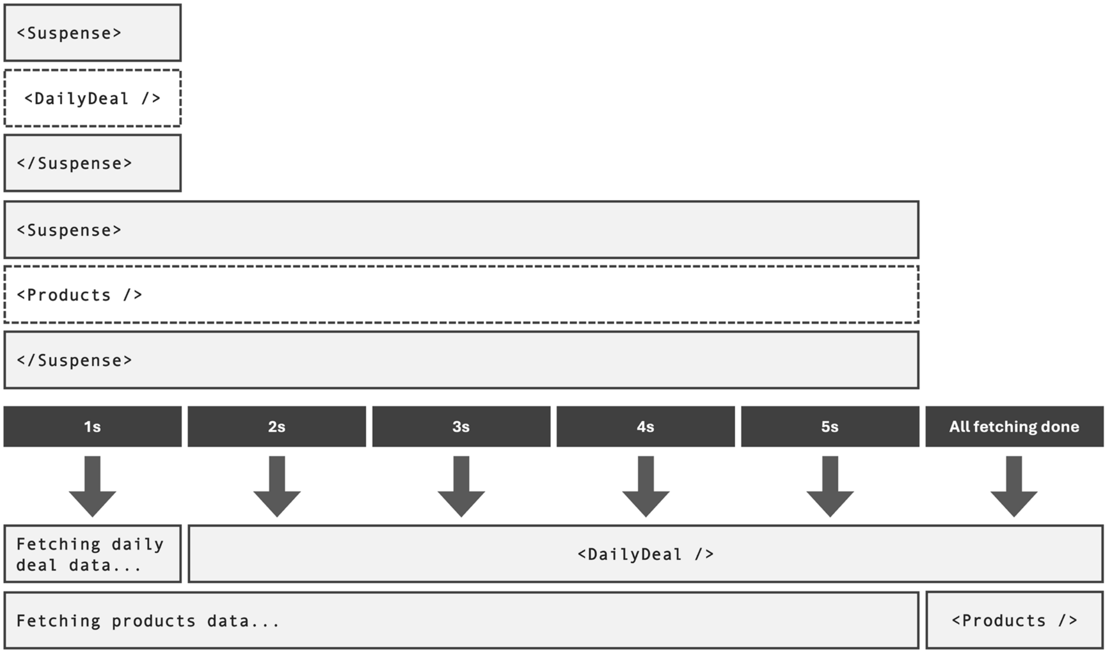
Revealing Content as Soon as Possible

Of course, there are situations where you might want to display fallback content for multiple components, but where you don't want to wait for all components to finish fetching data before showing any fetched content.

In such situations, you can use `Suspense` multiple times:

```
function Shop() {
  return (
    <>
      <h1>Welcome to our shop!</h1>
      <Suspense fallback={<p>Fetching daily deal data...</p>}>
        <DailyDeal />
      </Suspense>
      <Suspense fallback={<p>Fetching products data...</p>}>
        <Products />
      </Suspense>
    </>
  );
}
```

In this adjusted code example, `DailyDeal` and `Products` are wrapped with two different instances of the `Suspense` component. Thus, each component's content will be revealed once available, independent from the other component's data fetching status.


<Suspense>
<DailyDeal />
</Suspense>
<Suspense>
<Products />
</Suspense>
1s
2s
3s
4s
5s
All fetching done
Fetching daily deal data...
<DailyDeal />
Fetching products data...
<Products />

Nesting Suspended Content

Besides fetching in parallel, you can also create more complex loading sequences with nested `Suspense` components.

Consider this example:

```
function Shop() {
  return (
    <>
      <h1>Welcome to our shop!</h1>
      <Suspense fallback={<p>Fetching shop data...</p>}>
        <DailyDeal />
        <Suspense fallback={<p>Fetching products data...</p>}>
          <Products />
        </Suspense>
      </Suspense>
    </>
  );
}
```

In this case, initially, the paragraph with the text `Fetching shop data` is displayed. Behind the scenes, data fetching in the `DailyDeal` and `Products` components starts.

Once the `DailyDeal` component is done fetching data, its content is displayed. At the same time, below `DailyDeal`, the fallback of the nested `Suspense` block is rendered if the `Products` component is still fetching data.

Finally, once `Products` has received its data, the inner `Suspense` component's fallback content is removed, and the `Products` component is rendered instead.

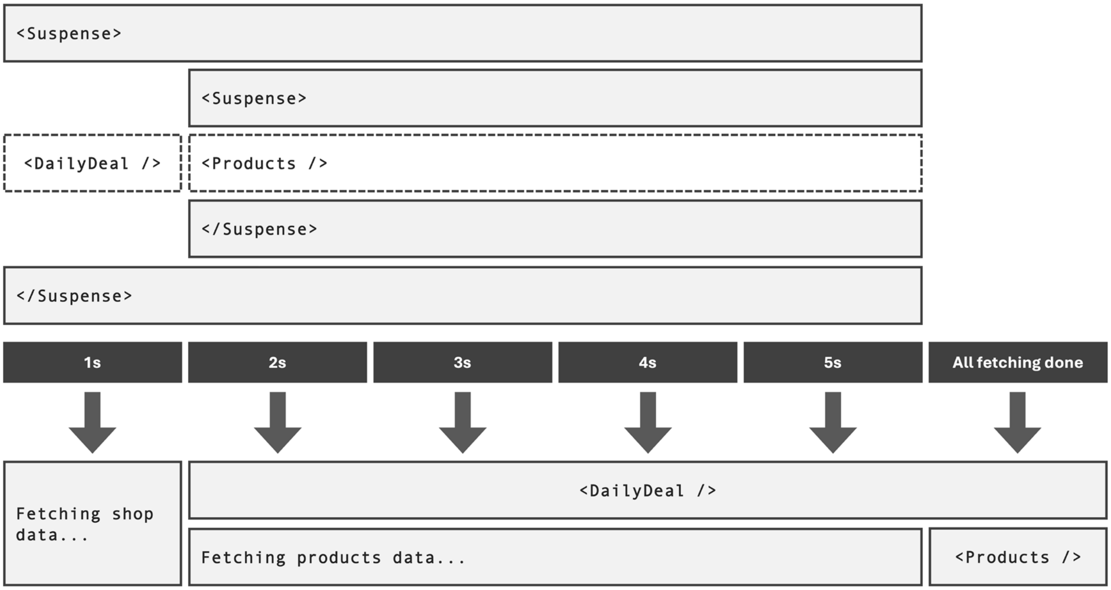


Figure 17.8: Nested Suspense blocks lead to sequential data fetching and content revelation

Therefore, as you can see, you can use `Suspense` multiple times. In addition, you can combine different `Suspense` components such that you can create exactly the loading sequence and user experience you need.

Should You Fetch Data via Suspense or useEffect()?

As you learned throughout this chapter, you can use `Suspense` in conjunction with RSCs, `Suspense`-enabled libraries, or the `use()` Hook (which also requires supporting libraries) to fetch data and show some fallback content while data is being fetched.

Alternatively, as covered in *Chapter 11*, *Working with Complex State*, you can also fetch data and manually show fallback content via `useEffect()` and `useState()` or `useReducer()`. In that case, you essentially manage the state that determines whether to show some loading fallback content on your own; with `Suspense`, React does that for you.

Consequently, it's up to you which approach you prefer. Using `Suspense` can save you quite a bit of code since you don't need to manage these different state slices manually. Combined with frameworks like Next.js or libraries like TanStack Query, data fetching can therefore become significantly easier than when doing it manually via `useEffect()`. In addition, `Suspense` integrates with RSCs and SSR and therefore can be used to fetch data on the server side—unlike `useEffect()`, which has no effect (no pun intended) on the server side.

However, if you're not using any library or framework that supports `Suspense` or `use()`-enabled promises, you don't have much of a choice other than to fall back to `useEffect()` (and hence not use `Suspense` for data fetching). This may change with future React versions, since they might provide tools that help with building promises that work with `use()`. But for the time being, it's basically a decision between using (the right) libraries and `Suspense` or no libraries and `useEffect()`.

Summary and Key Takeaways

- The `Suspense` component can be used to show fallback content while data is fetched, or code is downloaded.
- For data fetching, `Suspense` only works with components that fetch data via Suspense-enabled data sources during their rendering process.
- Libraries and frameworks like TanStack Query and Next.js support using `Suspense` for data fetching.
- Using Next.js, you can wrap `Suspense` around server components that use `async/await`.
- Alternatively, `Suspense` can be wrapped around components that use React's `use()` Hook for reading a promise value.
- `use()` should only be used to read values of promises that resolve with `Suspense` in mind—e.g., promises created by Suspense-compatible third-party libraries.
- When using Next.js, promises created in RSCs and passed to (client) components via props may be consumed via `use()`.
- The `use()` Hook helps with reading values and using `Suspense` in components that also need to use client-specific features like `useState()`.
- `Suspense` can be wrapped around as many components as needed to fetch data and display content simultaneously.
- `Suspense` can also be nested to create complex loading sequences.

What's Next?

React's `Suspense` feature can be very useful since it helps with granularly showing fallback content while code or data is being fetched. At the same time, when it comes to data fetching, it can be tricky to use `Suspense` since it only works with components that fetch data in the correct way (e.g., via the `use()` Hook, if the promise passed to the Hook is `Suspense`-compatible).

That's why this chapter also explored how to use `Suspense` and `use()` with Next.js, and how that framework simplifies the process of fetching data and showing fallback content with `Suspense` and `use()`.

Despite the potential complexity, `Suspense` can help with creating great user experiences since it allows you to easily show fallback content while a resource is pending.

This chapter also concludes the list of core React features you must know about as a React developer. Of course, you can always dive deeper to explore more patterns and third-party libraries. The next (and last) chapter will share some resources and possible next steps you could dive into after finishing this book.

Test Your Knowledge!

Test your knowledge of the concepts covered in this chapter by answering the following questions. You can then compare your answers to examples that can be found at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/blob/17-suspense-use/exercises/questions-answers.md`:

1. What's the purpose of React's `Suspense` component?
2. How do components need to fetch data in order to work with `Suspense`?
3. How may `Suspense` be used when working with Next.js?
4. What's the purpose of the `use()` Hook?
5. Which kind of promises can be read by the `use()` Hook?
6. List three ways of using `Suspense` with multiple components.

Apply What You Learned

With all the newly gained knowledge about Next.js, it's time to apply it to a real demo project.

In the following section, you'll find an activity that allows you to practice working with Next.js and `Suspense`. As always, you will also need to employ some of the concepts covered in earlier chapters.

Activity 17.1: Implement Suspense in the Mini Blog

In this activity, your job is to build upon the finished project from *Activity 16.1*. There, a very simple blog was built. Now, your task is to enhance this blog to show some fallback content while the list of blog posts or the details for an individual blog post are loading. To prove your knowledge, you should fetch data via `async/await` on the starting page (`/`), and via the `use()` Hook on the `blog/<some-id>` page.

In addition, the list of available blog posts should also be displayed below the details for a single blog post. Of course, while fetching that list data, some fallback text must be displayed—though, that text should be displayed independently from the fallback content for the blog post details.

Note

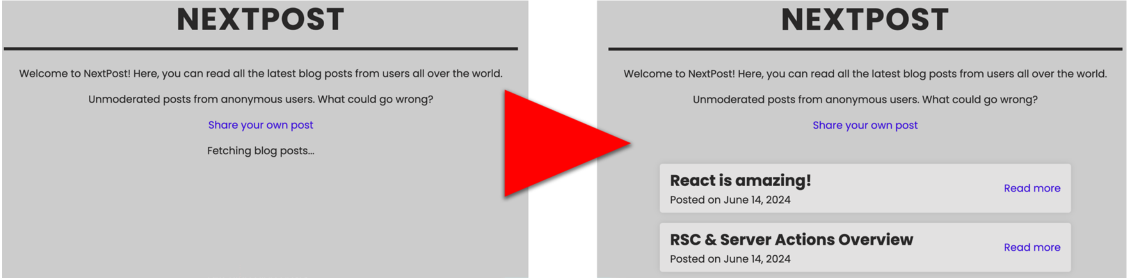
You can find a starting project snapshot for this activity at `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/17-suspense-use/activities/practice-1-start`. When downloading this code, you'll always download the entire repository. Make sure to then navigate to the subfolder with the starting code (`activities/practice-1-start`, in this case) to use the right code snapshot.

In the provided starting project, you'll find functions for fetching all blog posts and a single post. These functions contain artificial delays to simulate slow servers.

After downloading the code and running `npm install` in the project folder to install all required dependencies, the solution steps are as follows:

1. Outsource the logic for fetching and displaying a list of posts into a separate component.
2. Use that component on the starting page and use React's `Suspense` component to display some fitting fallback content while the blog posts are being fetched.
3. Also, outsource the logic for retrieving and rendering the details for a single blog post into a separate client (!) component. Output that newly created component on the `/blog/<some-id>` page.
4. Pass a promise for fetching the details of a blog to that newly created component, and use the `use()` Hook to read its value. Also, take advantage of the `Suspense` component to output some fallback content.
5. Re-use the component that fetches and renders a list of blog posts and output it below the blog post details on the `/blog/<some-id>` page. Use `Suspense` to show some fallback content, independently from the data fetching status of the blog post details.

The final page should look as shown in the following screenshots:



NEXTPOST

Welcome to NextPost! Here, you can read all the latest blog posts from users all over the world.

Unmoderated posts from anonymous users. What could go wrong?

Share your own post

Fetching blog posts...

NEXTPOST

Welcome to NextPost! Here, you can read all the latest blog posts from users all over the world.

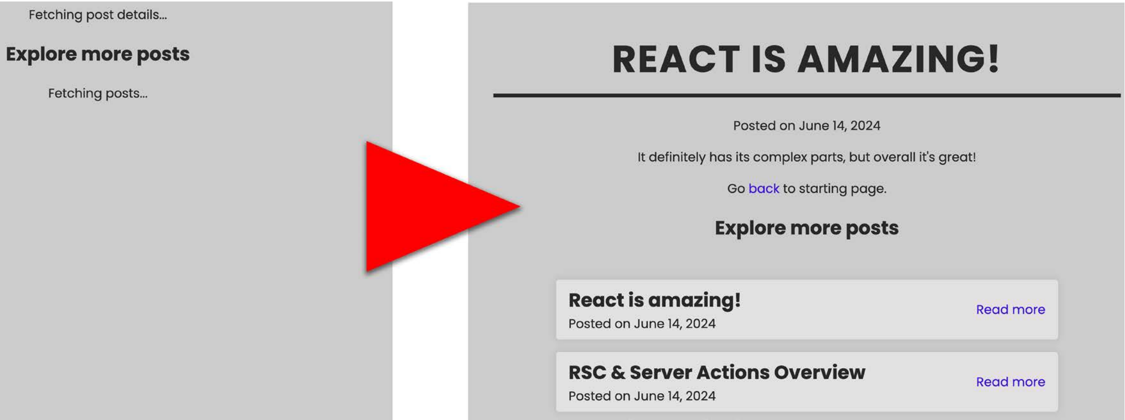
Unmoderated posts from anonymous users. What could go wrong?

Share your own post

React is amazing!
Posted on June 14, 2024
Read more

RSC & Server Actions Overview
Posted on June 14, 2024
Read more

Figure 17.9: Fallback content is shown while the blog posts are fetched



Fetching post details...

Explore more posts

Fetching posts...

REACT IS AMAZING!

Posted on June 14, 2024

It definitely has its complex parts, but overall it's great!

Go back to starting page.

Explore more posts

React is amazing!
Posted on June 14, 2024

RSC & Server Actions Overview
Posted on June 14, 2024

Figure 17.10: Fallback content is shown while fetching blog post details and the list of blog posts

Note

You can find the full code for this activity, and an example solution, here: `https://github.com/mschwarzmueller/book-react-key-concepts-e2/tree/17-suspense-use/activities/practice-1`.

18

Next Steps and Further Resources

Learning Objectives

By the end of this chapter, you will know the following:

- How to make the move from reading the book to applying your knowledge
- How to best practice what you've learned throughout this book
- Which React topics you can explore next
- Which popular third-party React packages might be worth a closer look

Introduction

With this book, you've gotten a thorough (re-)introduction to the key React concepts you must know in order to work with React successfully, providing both theoretical and practical guidance for components, props, state, context, React Hooks, routing, server-side React, and many other crucial concepts.

But React is more than just a collection of concepts and ideas. It powers an entire ecosystem of third-party libraries that help with many common React-specific problems. There is also a huge React community that shares solutions for common problems or popular patterns.

In this last, brief chapter, you'll learn about some of the most important and popular third-party libraries you might want to explore. You will also be introduced to other great resources that help with learning React. In addition, this chapter will share some recommendations on how best to proceed and continue to grow as a React developer after finishing this book.

How Should You Proceed?

Use the knowledge you gained throughout this book as a foundation to build upon. Dive deeper into Next.js, explore other popular React libraries, or learn more about React alternatives like Angular or Vue. Web development offers a broad range of technologies, languages, libraries, patterns, and concepts. And while this can sometimes feel overwhelming, it's also a vast pool of opportunities to grow as a developer and become better at solving complex problems.

But besides learning more about React and related packages, it's also important to apply your knowledge and practice what you've learned. Don't just read book after book. Instead, use your newly gained skills to build some demo projects.

You don't have to build the next Amazon or TikTok. There's a reason why applications like these are built by huge teams. But you should build small demo projects that focus on a couple of core problems. You could, for example, build a very basic website that allows users to store and view their daily goals, or build a basic Meetups page where visitors can organize and join meetup events.

To put it simply: practice is key. You must apply what you've learned and build stuff. Because by building demo projects, you'll automatically encounter problems that you'll have to solve without a solution at hand. You'll have to try out different approaches and search the internet for possible (partial) solutions. Ultimately, this is how you learn the most and how you develop your problem-solving skills.

You won't find a solution for all problems in this book, but this book does give you the basic tools and building blocks that will help you with those problems. Solutions are then built by combining these building blocks and by building upon the knowledge gathered throughout this book.

Become a Fullstack React Developer

This book already covered crucial concepts to get you started with React-based backend development. *Chapters 15*, *16*, and *17* explored server-side rendering, Next.js, server components and actions, and related features that will be needed to build fullstack React apps.

Consequently, diving deeper into Next.js might be an interesting next step. With the help of the official documentation or online courses like my *Next.js & React – The Complete Guide* course, you can acquire the necessary knowledge to become a fullstack React developer.

And it's not just Next.js: you can also explore alternatives like Remix and React Router (which is receiving more fullstack capabilities) or TanStack Start. If you don't care about having a fully integrated fullstack development experience as, for example, provided by Next.js, you can also learn more about connecting a decoupled backend to a React frontend—i.e., you can learn how to build and connect a separate backend (REST or GraphQL) API with Node.js or any other backend language.

Becoming a fullstack developer is not something you have to do, though. It's an option, but depending on your personal preferences or your role in a team, it might not be the right option for you. It's just important to know that building fullstack applications with React is one possible path you could explore—and that it's a path that became considerably easier with Next.js and similar frameworks. Either way, as mentioned before, you should also apply your React knowledge and practice by building demo projects, no matter whether you're diving deeper into fullstack development or not.

Interesting Problems to Explore

So, which problems and demo apps could you explore and try to build?

In general, you can try to build (simplified) clones of popular web apps (such as a highly simplified version of Amazon). Ultimately, your imagination is the limit, but in the following sections, you will find details and advice for three project ideas and the challenges that come with them.

Build a Shopping Cart

A very common type of website is an online shop. You can find online shops for all kinds of products—ranging from physical goods such as books, clothing, or furniture to digital products such as video games or movies—and building such an online shop would be an interesting project idea and challenge.

Of course, online shops do come with many features that can't be built with client-side React alone. For example, the whole payment process is mostly a backend task where requests must be handled by servers. Inventory management would be another feature that takes place in databases and on servers, and not in the browsers of your website visitors. Consequently, you can use Next.js (or one of the alternatives mentioned earlier in this chapter) to take care of this backend functionality and thus build a fullstack React application. But even if you don't want to dive into fullstack development, online shops contain many features that require interactive user interfaces (and, therefore, benefit from using React's client-side features). For example, you can set up different pages that show lists of available products, product details, or the current status of an order, as you learned in *Chapter 13*, *Multipage Apps with React Router*. You also typically have shopping carts on websites. Building such a cart, combined with the functionality of adding and removing items, would similarly utilize several React features—for example, state management, as explained in *Chapter 4*, *Working with Events and State*.

It all starts with having a couple of pages (routes) for dummy products (that are hardcoded into the frontend code and not fetched from some backend), product details, and the shopping cart itself. The shopping cart displays items that need to be managed via app-wide state (e.g., via context, as covered in *Chapter 11*, *Working with Complex State*), as website visitors must be able to add items to the cart from the product detail page. You will also need a broad variety of React components—many of which must be reusable (e.g., the individual shopping cart items that are displayed). Your knowledge of React components and props from *Chapter 2*, *Understanding React Components and JSX*, and *Chapter 3*, *Components and Props*, will help with that.

The shopping cart state is also a non-trivial state. A simple list of products typically won't do the trick—though you can, of course, at least apply your knowledge from *Chapter 5*, *Rendering Lists and Conditional Content*. Instead, you must check whether an item is already part of the cart or if it's added for the first time. If it's part of the cart already, you must update the quantity of the cart item. Of course, you'll also need to ensure that users are able to remove items from the cart or reduce the quantity of an item. And if you want to get even fancier, you can even simulate price changes that must be factored in when updating the shopping cart state.

As you can see, this extremely simple dummy online shop already offers quite a bit of complexity. Of course, as mentioned earlier, you could also add backend functionality and store dummy products in a database. If you want to, you can dive deeper into Next.js to build a more complex fullstack application based on React. This allows you to apply the knowledge you gained in *Chapter 15*, *Server-side Rendering & Building Fullstack Apps with Next.js*, and *Chapter 16*, *React Server Components & Server Actions*.

Build an Application's Authentication System (User Signup and Login)

A lot of websites allow users to sign up or log in. For many websites, user authentication is required before performing certain tasks. For example, you must create a Google account before uploading videos to YouTube or using Gmail (and many other Google services). Similarly, an account is typically needed before taking paid online courses or buying (digital) video games online. You also can't perform online banking without being logged in. And that's just a short list; many more examples could be added, but you get the idea. User authentication is required for a broad variety of reasons on many websites.

And on even more websites, it's optionally available. For example, you might be able to order products as a guest, but you benefit from extra advantages when creating an account (e.g., you may track your order history or collect reward points).

Of course, building your own version of YouTube is much too challenging to be a good practice project. There's a reason why Google has thousands of developers on its payroll. However, you can identify and clone individual features, such as user authentication.

Build your own user authentication system with React. Make sure that users can sign up and log in. Add a few example pages (routes) to your website and find a way of making some pages only available to logged-in users. These targets might not sound like much, but you will actually face quite a lot of challenges along the way—challenges that force you to find solutions for brand-new problems.

While you could just use some dummy (client-side) logic in your React app code to simulate HTTP requests that are sent to your servers behind the scenes, you could also add a real demo backend instead. That backend would need to store user accounts in a database, validate login requests, and send back authentication tokens that inform the React frontend about the current authentication status of a user. In your React app, these HTTP requests would be treated as side effects, as covered in *Chapter 8*, *Handling Side Effects*.

Again, if you want to use a real backend, you'll also need to dive into backend development and either build a separate server-side application or use Next.js (or any similar fullstack React framework). Alternatively, you can also use services like Firebase, Supabase, Auth0, or one of the many other services that provide authentication backends for frontend applications. Either way, you can explore how to connect your React app to such a backend.

As you can tell, this "simple" project idea (or, rather, feature idea) presents a lot of challenges and will require you to build on your React knowledge and find solutions for a broad variety of problems.

Build an Event Management Website

If you first were to build your own shopping cart system and get started with user authentication, you could then take it a step further and build a more complex website that combines these features (and offers new, additional features).

One such project idea would be an event management site. This is a website on which users can create accounts and, once they're logged in, events. All visitors can then browse these events and register for them. It would be up to you whether registration as a guest (without creating an account first) is possible or not.

It's also your choice whether you want to add backend logic (that is, a server that handles requests and stores users and events in a database) or you will simply store all data in your React application (via the app-wide state). If you don't add a backend, all data will be lost whenever the page is reloaded, and you can't see the events created by other users on other machines, but you can still practice all these key React features.

There are many React features that are needed for this kind of dummy website: reusable components, pages (routes), component-specific and app-wide state, handling and validating user input, displaying conditional and list data, and much more.

Again, this is clearly not an exhaustive list of examples. You can build whatever you want. Be creative and experiment because you'll only master React if you use it to solve problems.

Common and Popular React Libraries

No matter which kind of React app you're building, you'll encounter many problems and challenges along the way. From handling and validating user input to sending HTTP requests, complex applications come with many challenges.

You can solve all challenges on your own and even write all the (React) code that's needed on your own. And for practicing, this might indeed be a good idea. But as you're building more and more complex apps, it might make sense to outsource certain problems.

Thankfully, React features a rich and vibrant ecosystem that offers third-party packages that solve all kinds of common problems. Here's a brief, non-exhaustive list of popular third-party libraries that might be helpful:

- **TanStack Query**: A very popular library that helps with data fetching, caching, and management in React apps (`https://tanstack.com/query/latest`).
- **Framer Motion**: A React-specific library that allows you to build and implement powerful, visually pleasing animations into your React apps (`https://www.framer.com/motion/`).
- **React Hook Form**: A library that simplifies the process of handling and validating user input (`https://react-hook-form.com/`).
- **Formik**: Another popular library that helps with form input handling and validation (`https://formik.org/`).
- **Axios**: A general JavaScript library that simplifies the process of sending HTTP requests and handling responses (`https://axios-http.com/`).
- **Redux**: In the past, this was an essential React library. Nowadays, it can still be important as it can greatly simplify the management of (complex) cross-component or app-wide state (`https://redux.js.org/`).
- **Zustand**: If you are in need of an extra library that helps with managing state in React apps, you can also explore Zustand—a very popular alternative to Redux (`https://zustand-demo.pmnd.rs/`).

This is just a short list of some helpful and popular libraries. Since there's an endless number of potential challenges, you could also compile an infinite list of libraries. Search engines and Stack Overflow (a message board for developers) are your friends when it comes to finding more libraries that solve other problems.

Using TypeScript

You may also consider using TypeScript, instead of plain JavaScript, for your React projects.

TypeScript is a JavaScript superset that adds strong and strict typing. As a result, using TypeScript can help you catch and avoid certain errors related to missing values or incorrect value types.

You can get started with TypeScript for React with the help of the official documentation (`https://react.dev/learn/typescript`) or dedicated online courses or tutorials.

Other Resources

As mentioned, React does have a highly vibrant ecosystem—and not just when it comes to third-party libraries. You'll also find thousands of blog posts, discussing all kinds of best practices, patterns, ideas, and solutions to possible problems. Searching for the right keywords (such as *React form validation with Hooks*) will almost always yield interesting articles or helpful libraries.

You'll also find plenty of paid online courses, such as the *React – The Complete Guide* course at `https://www.udemy.com/course/react-the-complete-guide-incl-redux/`, and free tutorials on YouTube.

The official documentation is another great place to explore as it contains deep dives into core topics as well as more tutorial articles: `https://react.dev/`.

Beyond React for Web Applications

This book focused on using React to build websites. This was for a couple of reasons. The first is that React, historically, was created to simplify the process of building complex web user interfaces, and React is powering more and more websites every day. It's one of the most widely used client-side web development libraries and is more popular than ever before.

But it also makes sense to learn how to use React for web development because you need no extra tools—only a text editor and a browser.

That said, React can be used to build user interfaces outside the browser and websites as well. With React Native and Ionic for React, you have two very popular projects and libraries that use React to build native mobile apps for iOS and Android.

Therefore, after learning all these React essentials, it makes a lot of sense to also explore these projects. Pick up some React Native or Ionic courses (or use the official documentation) to learn how you can use all the React concepts covered in this book to build real native mobile apps that can be distributed through the platform app stores.

React can be used to build all kinds of interactive user interfaces for various platforms. Now that you've finished this book, you have the tools you need to build your next project with React—no matter which platform it targets.

Final Words

With all the concepts discussed throughout this book, as well as the extra resources and starting points to dive deeper, you are well prepared to build feature-rich and highly user-friendly web applications with React.

No matter if it's a simple blog or a complex Software-as-a-Service solution, you now know the key React concepts you need in order to build a React-driven web app your users will love.

I hope you got a lot out of this book. Please share any feedback you have, for example, via X (`@maxedapps`) or by sending an email to `customercare@packt.com`.

Join Us on Discord

Read this book alongside other users, AI experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the author via Ask Me Anything sessions, and much more.

Scan the QR code or visit the link to join the community.

`https://packt.link/ReactKeyConcepts2e`

`packt.com`

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

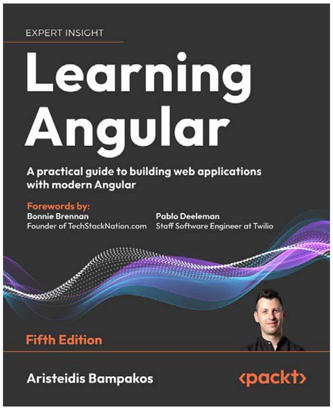
Why Subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At `www.packt.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

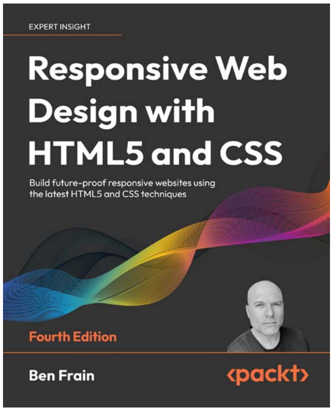


Learning Angular

Aristeidis Bampakos

ISBN: 9781835087480

- Use the Angular CLI to scaffold, build, and deploy new Angular applications
- Create Angular applications using standalone APIs
- Build rich components with Angular template syntax
- Apply reactivity patterns with the RxJS library and Signals
- Craft beautiful user interfaces using Angular Material
- Create HTTP data services to access APIs and provide data to components
- Improve your debugging and error handling skills during runtime and development
- Optimize application performance with SSR and hydration techniques

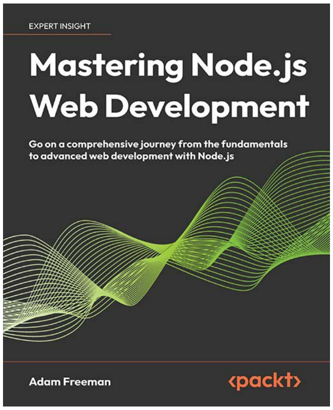


Responsive Web Design with HTML5 and CSS

Ben Frain None

ISBN: 9781803242712

- Use media queries, including detection for touch/mouse and color preference
- Learn HTML semantics and author accessible markup
- Facilitate different images depending on screen size or resolution
- Write the latest color functions, mix colors, and choose the most accessible ones
- Use SVGs in designs to provide resolution-independent images
- Create and use CSS custom properties, making use of new CSS functions including ‘clamp’, ‘min’, and ‘max’
- Add validation and interface elements to HTML forms
- Enhance interface elements with filters, shadows, and animations



Mastering Node.js Web Development

Adam Freeman

ISBN: 9781804615072

- Process HTTP requests and perform file operations
- Create RESTful web services that can be consumed by client-side apps
- Work with server apps serving JavaScript clients, such as React and Angular
- Leverage Node.js to work with popular databases
- Apply practical knowledge through building the SportsStore project
- Authenticate users and authorize access to application features

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *React Key Concepts, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

actions 207
  dispatching 293-296
advanced data fetching
  with Next.js 451
application programming interface (API) 174
App Router 408, 409
array destructuring 64
async() function
  reference link 175
asynchronous actions
  versus synchronous actions 216
authentication system
  building 496
await() function
  reference link 175
Axios
  reference link 497

B

Bootstrap CSS framework 137
built-in components 26, 27
built-in request interface
  reference link 368

C

Cascading Style Sheets (CSS) 117
children property 46, 47
class-based components 21
client actions 214
client components 433
  versus React Server Components (RSC)s 440
client-side React 440-442
client-side React apps
  disadvantages 402
  issues 402
code debugging 262-265
component evaluations 236-238
component functions 22
  calling 238
  naming conventions 27, 28
components 18, 43
  anatomy 19-21
  built-in components 26, 27
  need for 18
  props, consuming in 45, 46
  props, passing to 44
  props, using in 44
  selecting, for props 47
  significance 43, 44
  splitting 37

Component Updates 236-238

conditional content 90
  examples 90

conditional styles 126

conditional ternary operators 94

content
  rendering, conditionally 90-93

context access
  custom Hooks, using 321, 322

context API
  changing, from nested components 283, 284
  code completion 285
  context logic, outsourcing 286, 287
  context values, managing 276-281
  context values, providing 276-281
  lift state up 285
  multiple context, combining 287
  useState(), limitations 288-291
  using 284
  using, in nested components 281, 282
  using, to handle multi-component state 275, 276

controlled components
  versus uncontrolled components 160-163

Cross-Component State
  problem 272-275

CSS-in-JS solution 135

CSS styling frameworks 140

CSS styling libraries 140

custom components
  Refs 154-160

custom Hooks 301-308
  building 303, 305
  flexible feature 308
  parameters 309
  return values 310, 311
  using, for context access 320, 322

custom Hooks, example 312, 314
  first version, building 314, 315
  return values 316, 317
  reusability, improving 317-320

D

data
  loading, for Dynamic Routes 366, 367
  loading, with React Router 361-363

data entity 37

data fetching 360, 389-393
  via Suspense 487
  via useEffect() 487

data loading 393-395

data mutations 453
  handling, with Server Actions 453

data submission 376-379, 389-393
  <Form> triggers 385
  action(), working with 380-383
  current navigation status 386, 387
  data returning 383, 385
  Form Data, working with 380-383
  Forms, submitting 388, 389

declarative code
  used, by React 6-9

dependencies function 182, 183
  functions as dependencies 189-194
  unnecessary dependencies 183-185

directive 418

Document Object Model (DOM) 4, 121, 145
  accessing, with Refs 151-154
  manipulating, with React 9, 10
  reference link 4

DOM API 4

dynamic content 34
  outputting 34

Dynamic Routes 343, 344
  dynamic links, creating 346, 347
  programmatic navigation 348-351
  route parameters, extracting 345, 346
  used, for loading data 366, 367
dynamic styles 126

E

effect function 182, 183
  after effects, cleaning up 185-188
  asynchronous code 201, 202
  multiple effects, dealing with 189
  rules of Hooks 202
  unnecessary effect execution, avoiding 194-200
EJS (Embedded JavaScript templates) 19
element tags
  setting, conditionally 98, 99
entry point 23
error handling 374, 376
event management website
  building 496

F

fetcher object 391
  load() 391
  submit() 391
fetch() function
  reference link 175
first-class objects 25
fixed configuration options
  customizing 130, 131
form actions 80, 207, 214
Formik
  reference link 497
form submissions state
  managing 219
  pending UI state, handling with useFormStatus() 224, 225
  UI state, updating with useActionState() 219-221
form submissions with actions
  handling 214, 215
  synchronous actions, versus asynchronous actions 215, 216
form submissions without actions
  handling 208
  solution 213, 214
  user input, extracting 208
Framer Motion
  reference link 497
fullstack React developer 494
functional components 21

G

granular fallback content
  displaying, with Suspense 468, 469

H

Hooks 21, 63
  rules 203
HTML 28
HTML element style property
  reference link 122
HTTP requests 178, 179
  sending, without React Router 361

I

images
  rendering 35, 36
integrated development environment (IDE) 285
internal data 62

J

JavaScript CSS property names
  reference link 122
JavaScript Minifier Tool
  reference link 5
JavaScript styling frameworks 140
JavaScript styling libraries 140
JSX 6, 28-30
  React without JSX, using 30, 31
JSX elements
  closing tag 33
  regular JavaScript values 31-33

K

keys 109, 110

L

layout routes 335, 368, 371
  data, reusing 372-374
  example 369
list data
  examples 90
  mapping 102, 104
  outputting 100-102
list items
  issue 106-109
lists
  updating 104, 105
loader()
  params property 367
  request property 367
loader() function 363

M

memo() function 249
multiple props
  dealing with 48, 49
multiple state values
  merged state objects, managing 69-71
  state slices, using 68, 69
  state update, based on previous state 71-74
  two-way binding 75, 76
  working with 67

N

nested routes 335
Next.js 407, 408
  dynamic routes, handling 420-423
  exploring 424
  file-based routes, working with 410, 411
  filename conventions 424
  internal navigation, managing 415
  layouts, working with 412-414
  loading.js files, defining 451-453
  Next.js 11
  regular components, creating 418-420
  Suspense for data fetching, using 469-472
  used, for advanced data fetching 451
  used, for server-side rendering 411, 412
Next.js, internal navigation
  active links, highlighting 415-417
  managing 415
  use client directive, using 415-417
Next.js projects
  creating 408, 409
Node.js (LTS)
  reference link 11
normal component
  versus route component 341, 342
npm 1
  reference link 13

O

object destructuring 49

optimistic updates
  performing 226-230

P

params property 367

PascalCase naming convention 27

pending UI state
  handling, with useFormStatus() 224, 225
  managing, with useActionState() 222-224

portals 163-165
  using, for solving issue 165, 167

pre-processing/transpilation 6

prop chains 51

prop drilling 51, 274
  limitations 274

props 37, 46
  children property 46, 47
  components, selecting for 47
  consuming, in component 45, 46
  passing, to components 44
  spreading 49-51
  using, in components 44

R

React 1, 2, 65, 66
  Allowed State Value Types 67
  DOM, manipulating 9, 10
  naming conventions 66, 67
  reference link 2
  using, declarative code 6-9
  using, without JSX 30, 31

React apps
  problem 174-176

React apps styling
  components, building with customizable styles 129, 130
  conditional styles 126, 127
  CSS class styles, setting 123, 124
  dynamic styles, setting 124-126
  inline styles, using 121, 122
  multiple dynamic CSS classes, combining 127-129
  multiple inline style objects, merging 129
  working 118-121

React Compiler
  reference link 254

React Developer Tools 262-265

React feature
  Portals 163-165

React for Web Applications 498

React Hook Form
  reference link 497

React Hooks 21

React.js 1

React libraries 497

React Project
  creating, with Vite 11-13

React Query 476

React Router 326-329
  client-side code 368
  data load, accessing 364, 365
  data, loading for dynamic routes 366, 367
  layouts & nested routes, working with 334-338
  loaders 367, 368
  NavLink component, using 338-340
  package 326
  page navigation, adding 329-333
  requests 367, 368
  route component, versus normal component 340, 342
  used, for loading data 361-363

React Server Components (RSC)s 175, 407, 425, 429, 430
  advantages 432
  client components, combining 444
  client components, using in server components 445, 446
  components, building 431, 432
  creating 433
  project, setting up 438, 439
  server components, rendering via props 449, 450
  server components, using in client component 447, 448
  unlocking, in React Projects 433-438
  using 433
  versus client components 440
  versus server-side rendering (SSR) 439, 440

Real DOM
  versus virtual DOM 239-241

redirection 351
  lazy loading 352-354
  Undefined Routes, handling 352

reducer function 291, 293

Redux
  reference link 497

Refs
  controlled component, versus uncontrolled component 160-163
  in custom components 154-160
  usage, considerations 146-148
  using, for accessing DOM elements 151-154
  versus state 149-151

request property 367

resources 498

response interface
  reference link 364

rest property 50

root component 24

route component
  versus normal component 341, 342

routes
  defining 326-329

routing 10, 327, 360

S

scoped styles
  with CSS modules 132-134

server actions 214, 429, 453-459
  defining 454, 455
  project, setting up 438, 439
  storing, in separate files 460, 461
  triggering 454, 455
  unlocking, in React projects 453
  used, for handling data mutations 453

server-client boundaries 440

server-side data fetching
  problem 430

server-side rendering (SSR) 368, 403, 404, 429
  adding, to React application 404
  data fetching 405, 407
  versus React Server Components (RSC)s 439, 440

shopping cart
  building 495

short-circuiting 97

side effects 176, 177
  dealing, with useEffect() Hook 179, 180
  reference link 176

simple calculator
  building 85, 86
  enhancing 86, 87

single-page applications (SPAs) 10, 11, 119, 326, 402

spread operator 50, 129

state
  updating 62, 63
  useState() 63, 64
state batching 241, 242
state management
  with useReducer() 291
state slices 68
state values
  form submission, working with 79, 80
  forms, working with 79, 80
  lifting up 81-84
  using 76-78
static content 34
strict mode 261
  reference link 23
styled-components library 135-137
  reference link 137
Suspense for data fetching 488
  used, for displaying Granular Fallback Content 468, 469
  with Next.js 469-472
Suspense in React Projects
  data sources, setting up 476
  rendering process 474, 475
  Suspense for data fetching, using with support libraries 476-478
  use() 478-480
  useEffect() 473, 474
  using 472
Suspense usage patterns 483
  content, nesting 486
  content, revealing 484, 485
synchronous actions
  versus asynchronous actions 216
syntactical sugar 27

T

tagged templates 135
Tailwind CSS library for styling 137-139
TanStack Query 476
  reference link 497
techniques, of rendering content conditionally 94-98
  JavaScript logical operators, abusing 96, 97
  ternary expressions, using 94-96
template literal 348
ternary expressions 94, 96
transitions 217, 218
transpilation process 120
two-way binding 75, 76
TypeScript
  reference link 498
  using 498

U

UI state
  updating, with useActionState() 219-221
uncontrolled components
  versus controlled components 160-163
uniform resource locators (URLs) 325
Unnecessary Code Download
  avoiding 255
  lazy loading 255-261
use() 478-480
  in client components 482, 483
  with Next.js promises 480-482
useActionState() 457-459
  used, for managing Pending UI state 222-224
  used, for updating UI state 219-221
use client directive 442-444

useEffect()
  used, for data fetching 488
  used, for dealing side effects 179, 180
  using 180, 181
useFormStatus()
  used, for handling Pending UI state 224, 225
useLoaderData() 364
useMemo() 249
useOptimistic() 228
useParams() Hook 374
useReducer() 291
  actions, dispatching 293-296
  reducer function 291, 293
  used, for state management 291
User Input
  event object 211, 212
  extracting 208
  handling 455, 456
  refs 210, 211
  state, tracking 209, 210
user interface (UI)
  updating 455, 456
useRouteLoaderData() Hook 374

V

vanilla JavaScript 1, 28
  issues 2-5
virtual DOM 29
  costly computations, avoiding 247-250
  React compiler, using 253, 254
  state batching 241, 242
  unnecessary child component evaluations, avoiding 242-247
  useCallback(), utilizing 251-253
  versus real DOM 239, 240
Vite
  used, for creating React Project 11-13
Vite-based React router app
  migrating 426-428

W

web components 27
wrapper component 127

Z

Zustand
  reference link 497

Download the Free PDF and Supplementary Content

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

Additionally, with this book you get access to supplementary/bonus content for you to learn more. You can use this to add on to your learning journey on top of what you have in the book.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:

   `https://packt.link/supplementary-content-9781836202271`

2. Submit your proof of purchase.
3. Submit your book code. You can find the code on page no. 169 of the book.
4. That's it! We'll send your free PDF, supplementary content, and other benefits to your email directly

Description of Supplementary Content

This book comes with the following bonus material (claimable via the mechanism described above):

- A cheatsheet accompanying every chapter of the book
- A video in which author Maximilian gives you his recommendations for next steps after finishing this book
- A video in which author Maximilian shares his thoughts about the future of React